Scene decompositions for accelerated ray tracing

Spackman, John Neil

Award date:
1989

Awarding institution:
University of Bath

Scene Decompositions
for
Accelerated Ray Tracing

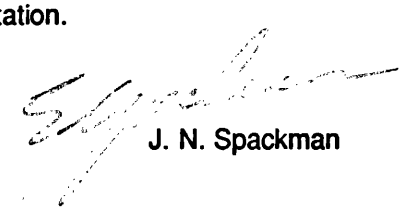
submitted by

John Neil Spackman

for the degree of Ph.D of the

University of Bath

1989



J. N. Spackman

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

To **Liz**

& Elaine

without whose support

this thesis would have finished here

Summary

Ray tracing has come to the fore in the computer synthesis of realistic images over the last decade. This algorithm synthesises a particularly high degree of realism in both the shading and shape of surfaces. Surface shading calculations incorporate not only local diffuse and specular radiance but also global shadowing, multiple reflection and refraction of view and light attenuation, all according to the laws of optical physics. Complex object shapes may be modelled as boolean constructs from exact specifications of many common geometries. A wide range of objects may be modelled exactly without resorting to polyhedral approximation.

Early implementations of ray tracing synthesised some of the most realistic images up to that date, but imposed an extremely high computational load for complex scenes. Synthesis times proved to be linear in object count, and projected times for scenes of a few thousand objects extended into months on popular mini-computers. This prevented the wide-spread application of ray tracing on non-specialised hardware. Various methods have been proposed over the last few years to improve the efficiency of ray tracing for more viable synthesis times.

This thesis addresses scene decompositions for the acceleration of ray traced image synthesis. The decomposition of a globally complicated scene into simpler local regions is shown to reduce the computational load imposed by ray tracing. Algorithms are presented for the construction and use of various types of scene decomposition. Their relative merits are compared and the 'octtree' decomposition is shown to be particularly suitable for accelerating the synthesis of complex scenes.

Contents

Preface

Synopsis:

> *This preface describes the structure of the thesis and its reference system. This should assist the reader in following references within the thesis and to other relevant publications.*

Reading this Thesis

This thesis addresses the efficient synthesis of realistic images by computer. Novel algorithms are presented which exploit scene decompositions to avoid the computationally expensive arithmetic of previous synthesis methods. An understanding of the associated mathematics is necessary to appreciate the extent of gains in efficiency. The thesis therefore consists of a narrative overview of image synthesis algorithms punctuated by detailed mathematical explanations.

Where possible such mathematics are referenced from the narrative text in separate figures, allowing the reader to follow the general chain of reasoning without becoming trapped in detail. The narrative is distinguished from figures by font. The narrative is in 'helvetica font':

Helvetica Font: Jackdaws love my big sphinx of quartz

Figures are printed in 'Roman font':

Roman Font: Jackdaws love my big sphinx of quartz

The Narrative

The narrative is structured by the Dewey digit system. For example, section two of chapter four is referenced as [Section 4.2]. Each chapter is preceded by a synopsis of its contents.

The Figures

Each figure is placed on the first available new page after the initial reference from the narrative. The position of any figure within the narrative is indicated by its reference. This

is a reference to the containing narrative section followed by a letter for unique identification. For example, the third figure in section two of chapter four would be [Fig 4.2c].

Geometric Notation

Linear Algebra is used extensively in realistic image synthesis. For example, linear transformations including projection and rotation are modelled with matrices acting on vectors. A standard notation is adopted for such algebra.

Vectors are denoted in underlined lower case: $\underline{v}$. The components of a vector are denoted with an appropriate subscript: $\underline{v} = (v_x, v_y, v_z)$. Vector dot product is denoted as $\underline{v}.\underline{u} = v_x u_x + v_y u_y + v_z u_z$ and cross product as $\underline{v} \times \underline{u} = (v_y u_z - v_z u_y, v_z u_x - v_x u_z, v_x u_y - v_y u_x)$.

Matrices are denoted in upper case: M. The column vectors of a given matrix are denoted as $M = [\underline{m_1}, \underline{m_2}, \underline{m_3}]$. Any matrix is taken to act on column vectors by pre-multiplication: $\underline{u} = M\underline{v} = \underline{m_1} v_x + \underline{m_2} v_y + \underline{m_3} v_z$.

References to Relevant Publications

Relevant publications are referenced by author and year. Where the same author has several relevant publications in the same year a letter is added for unique identification. A bibliography of all references is appended.

The Meaning of 'Realism'

The thesis addresses the synthesis of *realistic images* from a given scene model, rather than the generation of *realistic scene models*. The term 'realism' refers to the accuracy of synthesised images. This is judged in terms of surface shape for exact object specifications without resorting to polyhedral approximation, and of surface shading for exact surface normals allowing for various optical effects. The reader should not expect to be presented with specimen images of complex buildings or vehicles. This is due to limitations in the available scene models rather than the synthesis algorithms. The algorithms presented will synthesise realistic images from complex instances of a general scene model. This is verified by specimen images of scenes containing many objects with

various optical properties. For convenience these are arranged by procedural mathematics rather than according to realistic scenes. The synthesised realism would be the same in either case.

Acknowledgements

I would like to thank my supervisor Dr. P.J.Willis for his support and guidance during my research culminating in this thesis. This research has been funded by the SERC studentship 86315102 from 1986-1989. My thanks also go to many members of the Computing Group in the School of Mathematical Sciences for numerous useful discussions.

Chapter 1: Computer Image Synthesis

Synopsis:

Chapter one addresses the motivation for computer image synthesis. The development of this field is outlined up to the advent of ray tracing.

1.1. Motivation for Computer Image Synthesis

'One picture is worth more than ten thousand words'. Bartletts Familiar Quotations lists this well known saying as an anonymous Chinese proverb. The entry refers the reader to a similar quotation of the Russian author Turgenev - 'A picture shows me at a glance what it takes dozens of pages of a book to expound'. Neither has lost relevance in translation or time.

Pictures are a fundamental means of communication, transcending many barriers of language. They assist *visualisation, specification* and *simulation*. In the engineering industry, designs are visualised throughout their development with diagrams. The final design generally results in a draftsman's plans as a specification for manufacture. The twin tasks of designing a complicated vehicle and communicating the resultant specification to the manufacturer would be far less efficient in a purely textual format. Pictures also provide a means of simulation. A building's design is easily changed if the client recognises any adverse implications when presented with a pictorial simulation which is still 'on the drawing board', but less flexible once construction has begun. Vehicle pilot training currently makes extensive use of animated simulations. A trainee's error has far less serious consequences on a flight simulator than when flying a real aeroplane and allows early hands on experience. Animated simulation has also been used for special effects in films and scientific visualisation.

Images provide an indispensable aid to many applications since they are easily understood. However, the manual generation of these images is a more difficult task. Draftsmen and artists undertake skilled work which takes time. The delay is inconvenient when updating designs and renders real-time applications such as flight simulation impossible. This motivates image synthesis by computer for enhanced speed and flexibility.

1.2. The On Going Development of Computer Image Synthesis

Computer image synthesis has developed in parallel with hardware technology both for computation and display. Graphical displays first became available only a few decades

ago. These were vector devices only capable of drawing lines between screen points in a limited number of colours. The computational power available at the time for floating point intensive 3D geometry was limited. In its infancy computer image synthesis therefore addressed only simple *wire frame* object models. Such models are easily projected onto an arbitrarily placed view screen either with or without perspective. Lines along 3D edges in an object model are projected to 2D lines across the screen. Projection is therefore a linear operation and may be represented by a matrix. Early image synthesis exploited this linearity to restrict the expensive matrix computation of projection to object vertices. The projections of vertices linked by edges in the 3D model were then joined by lines in 2D screen space by the vector display hardware, requiring no further computation. Some enhancements were developed to this simple synthesis for extra realism. Where displays allowed, wire frame edges were depth-cued by contrast. The distance from the view screen to an object vertex is easily found with appropriate linear algebra. Hidden line algorithms were also developed. These synthesised images of objects modelled as polyhedral boundary approximations with black polygonal faces and white edges rather than wire frame models without faces. However, the limited capabilities of vector display technology precluded any attempts to shade such polygonal faces.

The opportunity to synthesise area shaded images arose with the advent of raster display devices. These display pictures as an array of picture element tiles or *pixels* in a rectangular arrangement. Each pixel is assigned a unit of memory whose contents define the colour displayed at the associated tile. Since many such discrete pixels are needed for an acceptable approximation to a continuous colour image, the raster device only entered into popular use as computer memory cost decreased. Raster displays may be driven similarly to vector displays with efficient 2D line generators [Bresenham;1965] to outline polygonal projections. Pixels in the interior of such projections may be set to an appropriate colour to synthesise surface shading. Whilst computational power improved with display technology, computation rates were still limited at the advent of the raster display. This necessitated the exploitation of image coherency to shade polygons at

reasonable speeds. Polygon shading was either in homogeneous colour or used incremental techniques to keep synthesis times feasible. In the simplest case polygons were assigned a constant colour, independent of surface orientation. More realistic shading was synthesised by allowing for this orientation, applying Lambert's law for diffuse shading and Phong's approximation for specular shading [Amanatides;1987]. However the faceted nature of polyhedral object approximations remained evident with homogeneous polygon shading. This was hidden by techniques developed by Gouraud and Bui-Tuong Phong to interpolate polygonal colour across the 2D raster screen [Gouraud;1971: Phong;1975]

The interpolation typically filled each polygon with incremental geometry, first between projected vertices along projected edges and then between projected edges across raster scan lines. Each object vertex was assigned a surface normal as an average over adjacent polygonal faces. Gouraud [1971] applied shading calculations at each vertex according to surface normal and interpolated colour directly within the polygon's interior. Phong [1975] interpolated surface normals to apply an independent shading calculation at each interior pixel. Interpolation proceeded incrementally to keep down computation costs. Facet artifacts have also been avoided by approximating object boundaries with local surface patches. Unlike polygons these join smoothly with continuous surface normals; bi-cubic patches are a typical example. Various algorithms have been developed to mask hidden surfaces such as the painter's algorithm, scan-line algorithms, Z buffers and Warnock's algorithm. These also generally attempt to exploit image coherency.

These methods synthesised far more realism than wire frame images. However, they synthesised images by projecting scene objects down onto the view screen. This supported only a *local* model of the interaction of light with scene objects before reaching the view screen - directly from light sources to a visible object surface and then to the screen. Since the interaction of light with other scene objects before reaching a visible surface was unknown, reflected and refracted views and shadows on object surfaces were difficult to synthesise. Some enhancements were made to synthesise shadows by visible surface calculations with respect to light sources rather than the viewer. Any surface obscured

from a light source was taken to be in shadow. Reflected and refracted views were at best approximated with environment texture maps [Amanatides;1987]. The local illumination model limited the realism of such synthesis.

Whilst techniques have been developed to hide the faceted nature of polyhedral object approximations in image synthesis, other problems presented by this model remain. A synthesised image is not always an end in itself. It may be an intermediary stage in some other process, for example providing visualisation during computer aided design (CAD) where the final goal is the actual manufacture of an object by computer aided manufacture (CAM). The graphical disguise of object facets does not carry over to computer controlled machining. Though modelled as polyhedral approximations, objects should be machined as exactly as possible. A more sophisticated object model and illumination model is desirable for many applications.

1.3. Ray Traced Image Synthesis

The on-going increase in computational power has made *ray traced* image synthesis feasible. Rather than projecting object silhouettes down onto a view screen, this synthesis traces light rays from the screen back into a scene. Extended interactions of each light ray with scene objects is modelled to support a *global* illumination model, allowing for multiple surface reflections of view, volume refractions of view, and surface shadowing [APPENDIX H]. The incremental geometry of polyhedral approximations may be abandoned to support a more complex object model. Objects are typically modelled from solids defined implicitly by polynomials. These model a wide range of objects without resorting to approximations.

Ray tracing rejects the computational savings achieved by previous techniques exploiting image coherency in favour of supporting sophisticated models for greater realism. This is not without computational cost. Chapter two describes the fundamental synthesis in detail. The specific cause of the high computational cost incurred by ray tracing is identified. This cost is shown to be catastrophically high for scenes containing more than a few objects, motivating research for more efficient strategies synthesising the same realism. This thesis is the culmination of such research.

Chapter 2: The Ray Tracing Algorithm

Synopsis:

Chapter two describes the synthesis of realistic images by ray tracing. This leads to an appreciation of the high computational cost involved, traditionally resulting in slow synthesis times. The cost is shown to be concentrated at a particular stage of synthesis. This bottleneck must be overcome to accelerate image synthesis.

2.1. Image Specification by Scene and View Models

Ray-tracing exploits the laws of optical physics to synthesise highly realistic images [Amanatides;1987: Whitted;1980: Appel;1970]. The application of these laws requires a suitably detailed description of the image to be synthesised. This is provided by two models. On one hand, the *scene* model specifies objects in terms of shape, position, orientation and constituent material [Fig 2.1a]. In an analogy to a film studio, the scene model describes the *set*. On the other hand, the *view* model specifies the viewing conditions, defining the viewer in terms of position, the view screen in terms of position, size and orientation, and the light sources in terms of position, intensity and colour [Fig 2.1b]. In the same analogy, the view model describes the *camera* and *lights* respectively.

The scene model deals with solid objects in a scene whilst the view model deals with light rays interacting with the scene. A raster image is ray traced pixel by pixel. The view screen is split into an array of rectangular tiles, each corresponding to a pixel in the synthesised image. Each pixel is assigned the colour observed by the viewer through the associated tile. This colour is found by tracing light rays backwards along their paths in three dimensional scene space - hence the algorithm's name. It depends on both the visible object surfaces, found by solution of the scene model, and the light reaching the viewer from these, found by solution of the view model [Fig 2.1c]. For purposes of display, colour is modelled as a three dimensional vector of intensities for the primary light frequencies corresponding to red, green and blue. Ideally, each intensity in the colour assigned to an indivisible pixel should be taken as the average over the associated view screen tile. This would allow for many visible objects contributing different colours over a single tile. Whilst such an average may be well defined as an integral over a tile's area, its evaluation is intractable for all but the most trivial cases. However, a standard statistical estimator such as sample mean is usually an acceptable approximation. A sample is taken of light intensity over a set of points within a tile. The sample points may be regularly distributed in a grid, or dispersed by some stochastic process [Cook et al;1984]. In the simplest case a single point sample is taken at a tile's centre. This is under the implied

Fig 2.1a: A Specimen Scene Model

An example of a textual scene model accepted as input for ray tracing:-

```
MATERIAL        China           Name of material to be specified
COLOUR          1 0.7 0.6       Colour as RGB triple
SURFACE         10              Sharpness of Specular Highlights
REFLECTION      0.5             Ratio of Specular to Diffuse Radiance
MIRROR          0               Proportion of Mirrored View
TRANSLUCENCY    0               Proportion of Refracted View
ATTEN_RATE      0               Attenuation on Light per Unit Length of Transmission
REFRACTION      1               Refractive Index

PRIMITIVE       Handle          Name of Primitive to be Specified
NAME            TORUS           Primitive Type
CENTRE          15 0 0          Coordinates of Centre
DEFORMATION     1  0  0         Deformation Matrix (90° Rotation About X Axis)
                0  0 -1
                0  1  0
SCALE           10 3            Major Radius of 10, Minor Radius of 3

PRIMITIVE       Inner           Name of Primitive to be Specified
NAME            CYLINDER        Primitive Type
CENTRE          0 1 0           Coordinates of Centre
DEFORMATION     1  0  0         Deformation Matrix (Identity)
                0  1  0
                0  0  1
SCALE           13 20           Radius 13, Height 20

PRIMITIVE       Outer           Name of Primitive to be Specified
NAME            CYLINDER        Primitive Type
CENTRE          0 0 0           Coordinates of Centre
DEFORMATION     1  0  0         Deformation Matrix (Identity)
                0  1  0
                0  0  1
SCALE           15 20           Radius 15, Height 20

OBJECT          ( Outer + Handle ) - Inner    Object Specified as CSG Construct
MADE_OF         China           Constituent Material
```

Fig 2.1b: A Specimen View Model

An example of a textual view model accepted as input for ray tracing:-

```
XMAX            256             Pixel Width of Image to be Synthesised
YMAX            256             Pixel Height of Image to be Synthesised
SCREEN          1 1 1           Screen Width, Height & Distance from Viewer
VIEW            0 30 -60        Coordinates of Viewer
SCENE_CENTRE    0 0 0           Coordinates of Centre of Interest
SCR_ROT         0               Screen Rotation from Upright
BCOLOUR         1 1 1           Background Colour as RGB triple

LIGHT           FINITE          Finite Light Source Specification
INTENSITY       2               Light Intensity
ORIGIN          -100 100 -100   Coordinates of Light Source
COLOUR          1 1 1           Light Colour

LIGHT           FINITE          Finite Light Source Specification
INTENSITY       1               Light Intensity
ORIGIN          100 100 -100    Coordinates of Light Source
COLOUR          1 1 1           Light Colour
```

Fig 2.1c: The Ray Tracing Algorithm

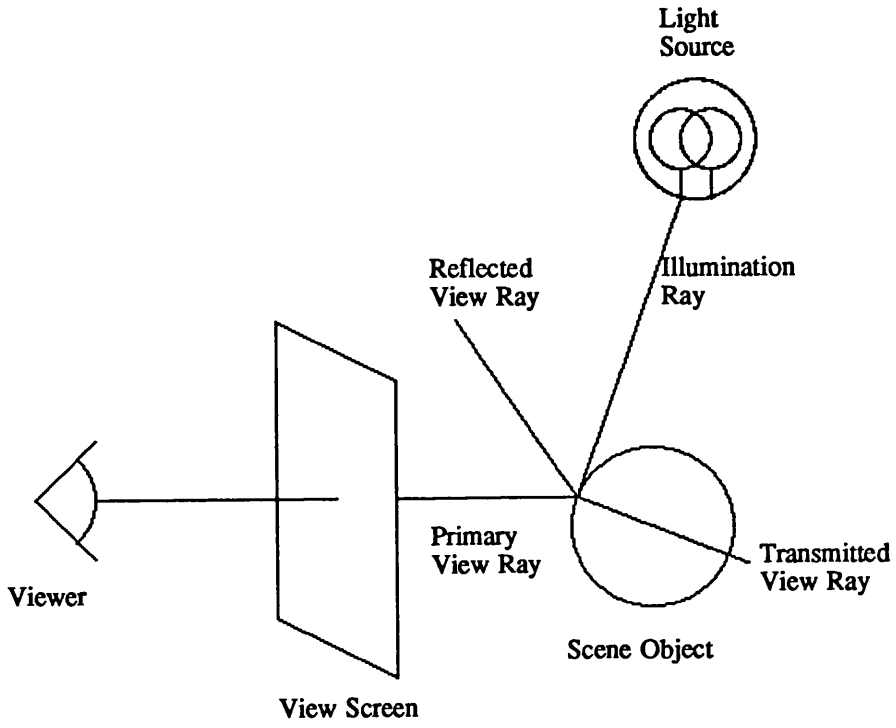


Fig 2.1c

assumption of colour being approximately constant over the tile, say with at most one visible object, so that a good estimate is obtained. This sampling method requires only the straightforward point evaluation of visible colour rather than integration over a two dimensional tile.

2.2. Visible Surface Calculations

Consider solving the scene model to find the visible surface through a single tile point. There is clearly at most one directly visible surface - the nearest, should this exist. This nearest surface is found by solving the scene model for a light ray starting at the viewer and extending through the point in question.

A ray is modelled as a half line by two vectors - the source position and unit direction [Fig 2.2a]. Any point on the one dimensional ray is then parameterised by path length from the source. The nearest surface is that yielding the non-empty ray intersection of minimal non-negative path length.

2.2.1. The Scene Model

The Constructive Solid Geometry (CSG) model is a common scene model [Roth;1982: Wyvill et al;1986]. This specifies a wide range of objects as binary boolean constructs from appropriately deformed instances of simple primitive solids. A range of common primitive geometries is modelled, typically comprising the planar bounded half-space, cube, sphere, cylinder, double cone and torus. Each is defined in a convenient local space centred at the local origin and oriented with the local axes. The choice of coordinate system is arbitrary. By convention, a left-handed system is taken throughout this thesis with increasing X going from left to right, Y from down to up and Z from behind to front. Each primitive has a size parameterisation within its local space, specifying aspects such as radius or height.

A primitive is instanced in the scene model by a transformation to the world coordinate system comprising any sequence of linear transformations such as translations, rotations, scalings, reflections and shears. Associated transformations such as the inverse to local space and surface normal transformations are derived from this [APPENDIX A].

Fig 2.2a: The Ray Model

The ray model is

$$\text{Ray}(\underline{s},\underline{d}) = \left\{ \underline{p} \in R^3 : \underline{p} = \underline{s} + \lambda \underline{d} \; ; \; \lambda \geq 0 \right\}$$

where
- $\underline{s}$ = ray source
- $\underline{d}$ = ray direction
- λ = path length

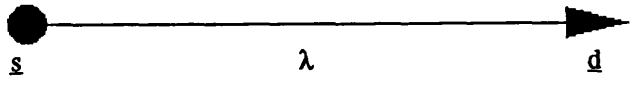


Fig 2.2a

Typical boolean operations are binary union, intersection, subtraction and symmetric difference together with unary identity, under the usual definitions.

A local cube is modelled as a suitable planar half space intersection in local space. The finite local cylinder and cone are modelled as the intersection of the infinite form with two clipping planar half-spaces. Any convex polyhedron is instanced in world space as a planar half space intersection. A concave polyhedron is instanced with extra boolean operations such as union. More complex curved objects are instanced with the full range of primitives, deformations and operations. CSG objects are conveniently described by their binary evaluation trees in which the internal nodes represent boolean operations and the leaf nodes represent primitive solids [Fig 2.2.1a]. A major advantage of this model is the exact specification of a wide range of curved objects without resorting to polyhedral approximations. The range of primitives may be augmented with volumes of revolution and extrusion, bi-cubic patch boundary representations and volume densities [Sederberg,Anderson;1984: Joy,Bhetanabhotla;1986: Burr;1986: Sweeny,Bartels;1986: Kajiya,Von Herzen;1984]. However the basic CSG model has proved general enough for many applications and is well established in fields such as computer aided design [Myers;1982]. Many objects can be modelled from a limited set of common primitives under suitable linear deformations and boolean combinations.

Each local primitive is modelled by a field function expressed as a trivariate polynomial in local space coordinates [APPENDIX B]. This specifies a notion of height above the primitive surface at any local point. This height may not be in a usual Euclidean sense, but will be zero for a point on the primitive surface, increase as this point is moved above the surface or decrease as it is moved below. The height function may be compared to the concept of height above sea level. The height above a primitive instance at a world scene point is taken as the local height value at that point's local image [APPENDIX A]. A scene point is on a primitive's surface when the associated world height function evaluates to zero, outside when the height is positive and inside when the height is negative.

Fig 2.2.1a: The Constructive Solid Geometry (CSG) Model

A mug constructed from two cylinders and a torus, as specified by Fig 2.1a.

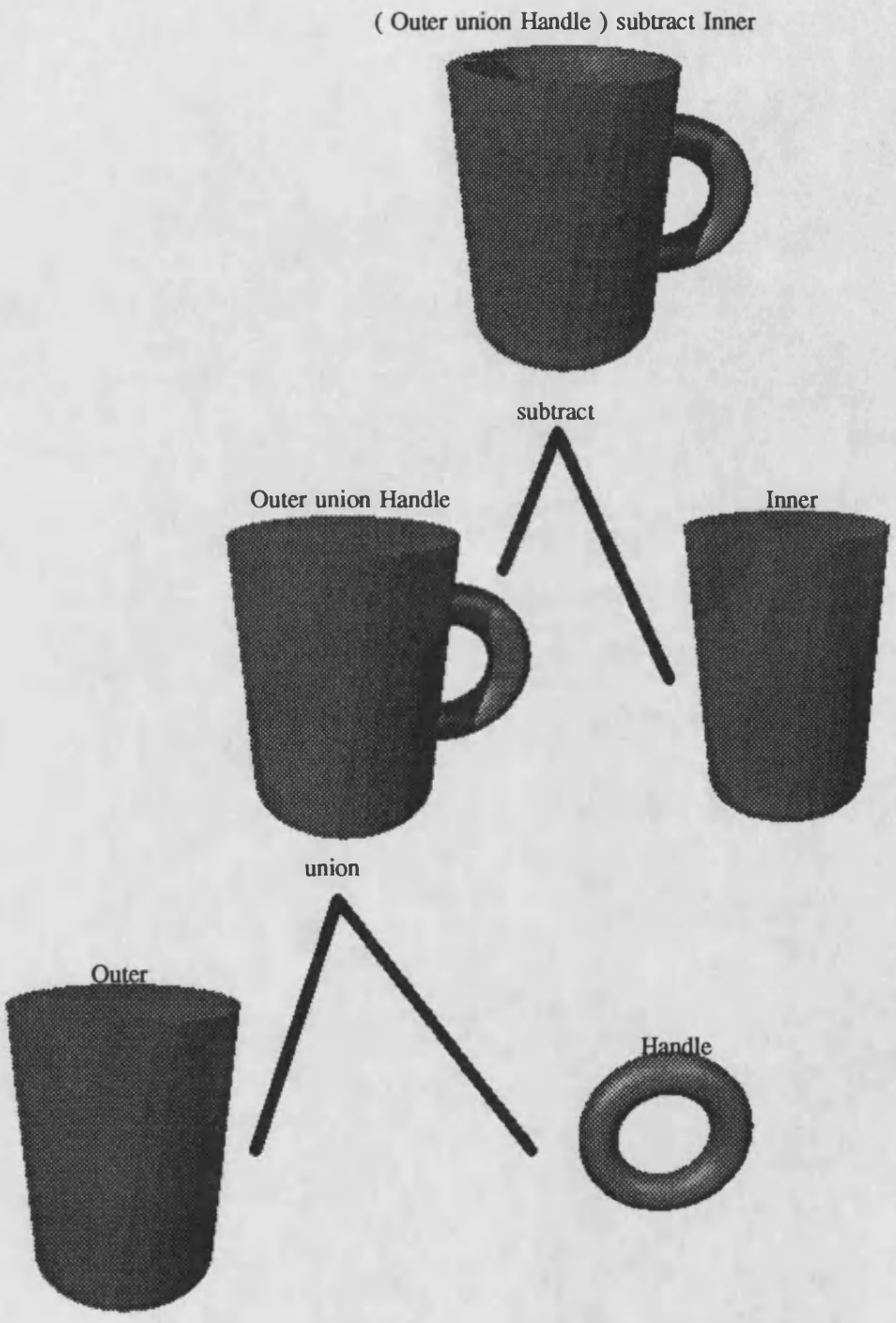


Fig 2.2.1a

2.2.2. The Solution of Scene Model Equations

A ray is intersected with a world primitive instance by the substitution of its local equation as the argument to the local height function. This yields univariate polynomials in the ray's path length [APPENDIX C]. Any intersection with the primitive's surface is then characterised by a root of these polynomials. This may be substituted back into the ray equation to find the visible surface point's position in either local or world space.

Height functions immediately generalise from primitives to CSG boolean constructs thereof and indeed to constructs of constructs. A height function is defined for a construct as piecewise segments of those already defined for the arguments [APPENDIX B]. A height function may then be recursively defined at each node in a CSG object's binary tree structure, yielding a height function for the whole object at the root. The substitution of the ray equation into these yields univariate continuous piecewise polynomials in the ray's path length [APPENDIX C].

The roots of a construct's height function may be found without calculating the explicit piecewise segments from the argument functions. Clearly, any root of the binary construct's height function must be inherited from one of the two arguments' height functions. A check on which roots of the arguments' simpler height functions remain roots of the construct height function is sufficient to locate *all* the latter's roots.

To look at the problem another way, consider finding a ray's intersection with an entire solid rather than just the surface boundary. This is expressed as the path length section over which the associated height function is negative. It does not consist of a few boundary points, but is rather the union of a finite number of path length intervals. Points on the boundary of this section correspond to ray intersections with the object surface as before. The path length section is distributive over all the boolean operations, in that *the path length section for a 3D combination of objects is the corresponding 1D combination of the objects' path length sections* [Fig 2.2.2a].

Such unions of a finite number of intervals are a convenient type for computation. They may be represented by a flag indicating whether the ray source is inside the associated

Fig 2.2.2a: Path Length Sections for Boolean Constructs

Let $r = \{ \underline{s}+\lambda\underline{d} : \lambda>0 \}$ be a ray, and $A,B \subseteq R^3$ be two constructs

A ray's path length section within any construct X is defined as $S_X = \{ \lambda > 0 : \underline{s}+\lambda\underline{d} \in X\}$

This definition is distributive over boolean operations. Consider the union of these constructs,

$$S_{A \text{ union } B} = \{ \lambda > 0 : \underline{s}+\lambda\underline{d} \in A \text{ union } B \}$$

$$= \{ \lambda > 0 : \underline{s}+\lambda\underline{d} \in A \} \text{ union } \{ \lambda > 0 : \underline{s}+\lambda\underline{d} \in B \}$$

So for union

$$S_{A \text{ union } B} = S_A \text{ union } S_B$$

Similarly,

$$S_{A \text{ intersect } B} = S_A \text{ intersect } S_B$$

$$S_{A \text{ subtract } B} = S_A \text{ subtract } S_B$$

$$S_{A \text{ difference } B} = S_A \text{ difference } S_B$$

construct, followed by a list of surface intersection points in path length order. Though this flag indicates whether the ray is initially inside or outside the construct, it is set according to the ray's final state. This avoids the impact of rounding errors in calculating the height function at any ray source which is close to the surface of a geometry. The source of a ray is taken to be inside a geometry if the leading coefficient of the height function along its path is positive and there is an odd root count, or if this coefficient is negative and there is an even root count. The source is taken to be outside the geometry otherwise. An implementation of binary boolean operations on this data type handles ray intersections with CSG objects by an elegant recursion over the associated binary construct tree structure. Spatial object coherency may be exploited to restrict computation to a path length sort over the arguments' surface intersection lists filtered by the appropriate boolean operators. The section for the entire object provides not only surface intersections but also the ray's path length within this object. This is needed to calculate the attenuation of a ray passing through a transparent object when solving the view model [APPENDIX G].

The intersection of a ray with the surface of a CSG object is calculated by isolating the roots of such polynomials. Roots may be found analytically for polynomials up to degree four [Korn,Korn;1968a], or with reliable numerical techniques for any degree [Korn,Korn;1968b]. The visible surface is the nearest to the ray source, on the object whose height function yields the minimal positive path length root.

2.2.3. The Efficiency of Visible Surface Calculations

The original method of solving the scene model for this minimal root is a computable but naive exhaustive search [Whitted;1980]. Each object is *queried* in turn for ray intersection whilst maintaining a record of the nearest intersected surface found to date. Having been initialised to an infinite root indicating that no visible surface has been found this record is updated whenever an object's surface is intersected by the ray at a shorter positive path length. If the record is still infinite after considering every object then no visible surface has been found and a background colour is assigned. This may be constant or textured [APPENDIX D]. Otherwise, the record indicates the nearest visible surface point, and the

visible colour may be found by solving the view model for the ray at this point. Each object query carries a considerable computational tariff. The roots of the associated height function are identified from those of the combined primitives. The location of roots to each primitive's height function involves costly multiplicative floating point arithmetic [Korn,Korn;1968a]. This is preceded by other floating point vector arithmetic in calculating the function's coefficients and transforming to local space [APPENDICES C,A]. Since every object is queried, the computational load of the scene model's solution by naive exhaustive search increases linearly with object count and becomes catastrophically expensive for any non-trivial count.

2.3. Shading Calculations

Consider solving the view model to find the visible colour reaching the viewer along a ray from a visible surface point. Light falling on a surface is called *irradiance* whilst that reflected back is called *radiance*. The view model's solution finds the irradiance falling on the viewer from the view direction due to radiance from a known visible surface point. Whilst a full colour image is synthesised, primary light frequencies are considered independently thereby simplifying the colour problem to monochrome over each of the three primaries. The radiant intensity of each primary is found by the laws of optical physics [Amanatides;1987].

2.3.1. The View Model

Surface radiance is summed over five categories corresponding to different types of irradiance and surface characteristics. These are

- Diffuse radiance

  due to irradiance falling directly from light sources. This results from the interaction of irradiance with matte surfaces such as porous chalk. Irradiant light penetrates such surfaces and is scattered through many internal reflections before re-emerging as diffuse radiance. This extended scattering distributes radiance uniformly in all directions according to Lambert's law, and attenuates light to the surface's

characteristic colour.

- Specular radiance

  also due to irradiance falling directly from light sources. This results from the interaction of light with mirrored or shiny surfaces to produce highlights. Light does not penetrate such surfaces, but reflects straight back according to Newton's law of reflection. Specular radiance is highly directional, but in practice is never restricted to a single direction since no actual surface is perfectly smooth. Real surfaces have many micro-facets, whose normals average out to the overall surface normal but are distributed with some variance. This is heuristically modelled by Phong's bell-shaped distribution [Amanatides;1987]. A more rigorous model has been proposed which is derived from Fresnel's laws [Amanatides;1987], but this offers only a limited increase in realism at higher expense. In the absence of any extended surface interaction, specular radiance is not attenuated according to surface colour in the view model.

- Ambient (diffuse) radiance

  due to irradiance which is not direct from light sources but radiant from other surfaces in the scene. A model has been presented to allow for this radiance and so synthesise optical effects such as caustics [Cook et al;1984]. However, this model is somewhat complicated and ambient radiance may be approximated with a constant term which proves adequate for many applications. Like diffuse radiance, ambient radiance is attenuated to the surface's characteristic colour.

- Reflected radiance

  from a mirror or refractive surface with total internal reflection, due to irradiance from the direction according to Newton's law. This carries a reflected view and like specular radiance is not attenuated to the surface colour.

- Transmitted radiance

  from a surface on a transparent refractive object without total internal reflection, due to irradiance from the direction according to Snell's Law. This carries a refracted view and is attenuated on transmission through the surface by characteristic colour.

2.3.2. The Solution of View Model Equations

The intensity of each radiance category depends on the viewing conditions according to optical laws. The direction in which the surface is viewed is known, as is the visible point. The surface normal may be found according to the surface's geometric type, negating when the surface is carved out by boolean complement in subtraction or symmetric difference [APPENDIX D]. The colour of the visible point may be constant across the surface, or textured according to local position [APPENDIX E]. Light sources are modelled as points. These may be at a finite distance from the surface at a known position, or at an infinite distance in a known direction. A unit direction vector towards the source is easily found in either case.

These viewing conditions are fed into the view model to find the total visible surface radiance as an average of the radiance categories weighted by surface characteristics [APPENDIX G]. The irradiance from each light source is required to calculate diffuse and specular radiance, as is that from the reflected view direction for reflected radiance and from the refracted view direction for transmitted radiance. Reflected and refracted irradiance are found by recursively tracing rays from the visible surface point in the appropriate directions, rather than from the viewer through the view screen. The appropriate directions are easily found [APPENDIX F]. The irradiance from each light source is also found by tracing a further ray. These are second generation or *secondary* rays, whereas those through the view screen are *primary* rays.

2.3.3. Synthesising Recursive Views

The reflected and refracted view rays are traced exactly as the primary view rays. The scene model is solved once more to find the nearest surface struck, as is the view model for shading. The latter may spawn yet further generations of rays, so that the final irradiance assigned as the visible colour along the primary ray is modelled as a branch-weighted average over a *shading tree* [Whitted;1980] in which each node represents a further visible surface [Fig 2.3.3a]. The tree terminates when a non-mirror, opaque surface is struck, or no surface at all. A background colour is assigned in the latter case. Some

Fig 2.3.3a: The Shade Tree

The final irradiance falling on the viewer is a weighted average over the branches of a *shade tree.*

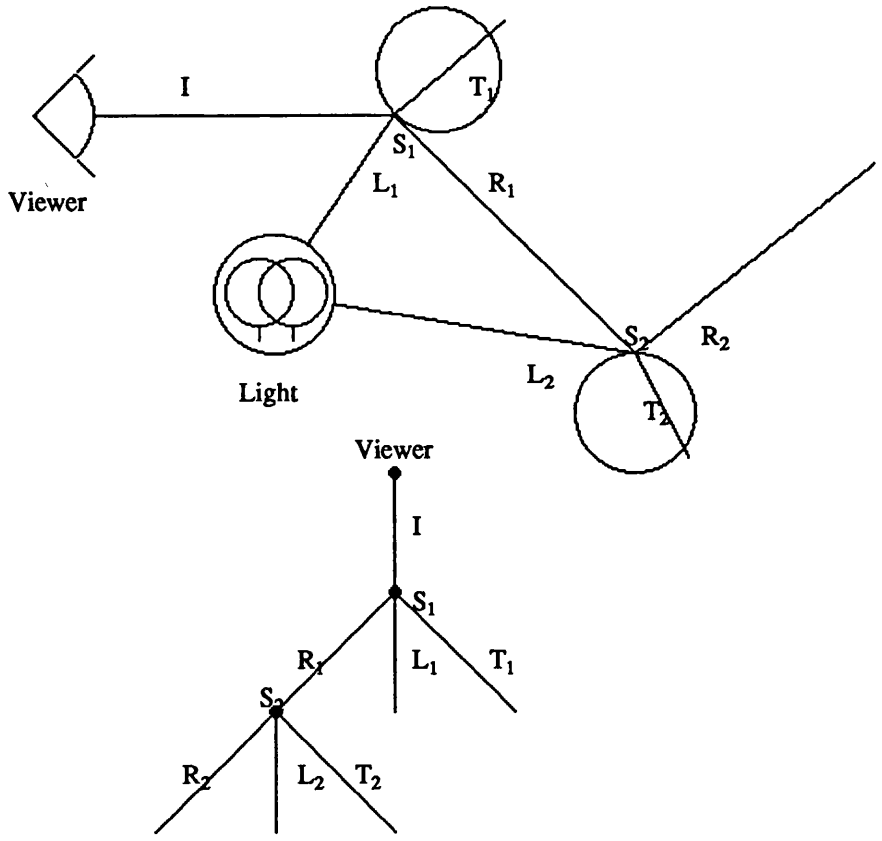


Fig 2.3.3a

branches of the shading tree may satisfy neither of these criteria, such as when a lineage of reflected view rays becomes trapped in a mirrored enclosure. Due to branch weighting however, the contribution to the root average of each successive generation will decrease, eventually decaying to an insignificant level. A maximum depth termination criterion is applied to the shading tree to prevent runaway. Any branch attempting to exceed this is pruned and assigned a background colour. This generally has little impact on the final image even when pruning only a few generations away from the root. The maximum depth is usually between five and ten. Pruning may also be applied to any branch contributing a lower proportion to the weighted root average than some minimum threshold, so avoiding unnecessary computation. The contribution of a branch is the product of all branch-weights over the path from the root, which is easily remembered when traversing over the tree. A minimum threshold of about one tenth is often used.

2.3.4. Synthesising Shadows

A ray traced to find the irradiance from a light source requires a slightly different solution of the scene model. This does not spawn further generations of rays. Such a ray is known as an *illumination* ray to distinguish it from the *view* rays discussed above. The scene model's solution for an illumination ray finds the proportion of radiance from a light source surviving the direct journey through the scene to fall as irradiance on the visible surface. The attenuation will depend on the various transmissive media encountered. If the ray strikes any opaque object, attenuation is complete and the object casts a shadow from the light over the surface point since this is not visible from the light source. If the ray strikes a transparent object however, a proportion of the light will be transmitted. Two types of attenuation are modelled in this case. The first is due to the filtering effect of the object's *surface* colour, whereby only each appropriate primary proportion is transmitted. This colour may be constant or textured by local position. The second is due to decay through the object's *body*. The proportion of light transmitted is calculated according to exponential decay [APPENDIX G]. This models a colourless transmissive medium, so that transparent objects are modelled as colourless bodies with a coloured surface somewhat like a

cellophane wrap of insignificant thickness. The same exponential decay model is used when viewing a surface through an attenuating medium to find the fraction of radiant light leaving the surface which reaches the viewer [APPENDIX G].

2.3.5. The Efficiency of Shadow Synthesis

An illumination ray's solution to the scene model must find not only the nearest object surface but all objects struck. These do not have to be considered in path length order, since the total proportion of light transmitted is modelled as the product over all objects, whose evaluation is of course commutative. Whilst view rays are traced from nearest surface to nearest surface, spawning a new generation each time, a single illumination ray is exhaustively traced from a visible point through the scene until reaching the light source. Again, the original method of the scene model's solution for an illumination ray is a computable but naive exhaustive search [Whitted;1980]. Each object is queried in turn for the proportion of light transmitted. Having been initialised to unity to indicate no attenuation, a record of the total proportion transmitted is maintained by multiplication with the fraction transmitted by each object struck. If this record is zero after considering all objects the surface point is in full shadow from this light source. Otherwise, some proportion of irradiance survives and is allowed for in the view model. Once again, each object query carries a considerable computational tariff. The exhaustive search can be aborted if this proportion record becomes zero before considering every object, as the surface must be in full shadow. However, this saving is only significant when a large proportion of visible surfaces are in full shadow.

2.4. The Computational Bottleneck

Whilst the original exhaustive search methods for solving the scene model are computable they are clearly naive. A high unit computational cost is incurred for both view and illumination rays, increasing linearly with object count. A realistically detailed scene many contain thousands of objects. Each is considered in an exhaustive search for a given ray.

This high cost could perhaps be ignored if the scene model were not solved very often. However, the solution of the scene model is at the core of the ray tracing algorithm [Fig 2.4a]. Consider synthesising an image at a typical spatial resolution of 1024×1024 pixels. Even for the simplest case of a single sample per pixel, visible surface calculations require the solution of the scene model for over one million primary view rays, and proportionately more if the sample size is increased. Moreover, shadowing calculations at each visible surface point struck by a primary ray require the scene model's solution for a secondary illumination ray to each light source. The majority of primary view rays will strike some surface for a busy or indoor scene. Many light sources may be needed to synthesise a realistic image, in which case the computation required for shadowing will often exceed that in finding visible surfaces. Any view ray striking a partially mirrored or transparent object will spawn further generations of view rays for which the scene model must be solved. These spawn more illumination rays in their turn, thereby increasing the number of solutions to the scene model yet further.

Ray tracing synthesises realism by quite literally solving the scene model millions of times. This high solution count is fundamental to the algorithm, providing the means for realistic synthesis. The high unit cost of traditional solution methods forms a catastrophic computational bottleneck resulting in notoriously slow rendering times. Thousands of objects may be needed to model a realistic image, resulting in literally billions of object queries. Research has shown that even for scenes containing only a few objects about three quarters of all computation time is spent solving the scene model [Whitted;1980]. When dealing with thousands of objects all but a tiny proportion is spent solving the scene model, with total synthesis times running into weeks and even months on non-specialised hardware.

This drawback of ray tracing has motivated research for a solution to the scene model of lower unit cost. The major part of this thesis addresses this task.

Fig 2.4a: Fundamental Flow Diagram for Ray Tracing

The solution of the scene model for view and illumination rays is at the core of image synthesis.

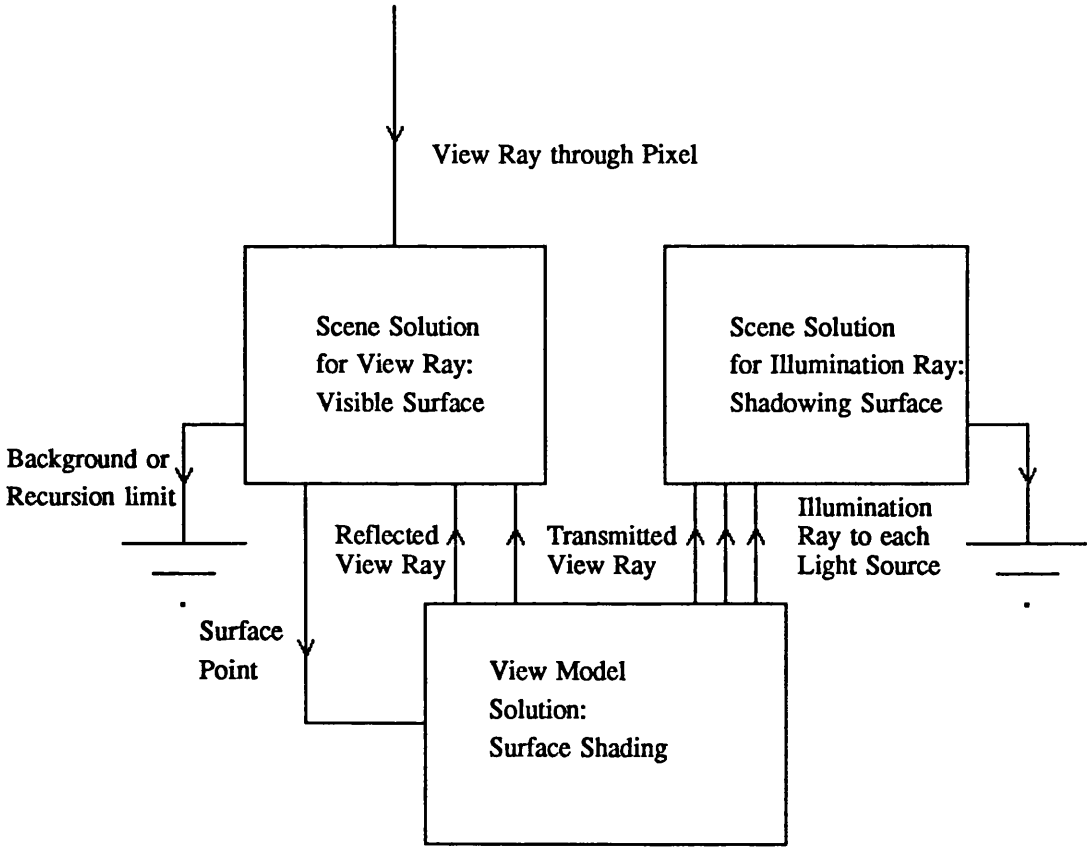


Fig 2.4a

Chapter 3: Ray Tracing Acceleration Techniques

Synopsis:

Chapter three describes how ray tracing may be accelerated for faster image synthesis. Fine tuning techniques and their effectiveness are addressed before more fundamental innovations which can radically improve speed.

3.1. Motivation for Accelerating Ray Traced Image Synthesis

Naive ray tracing has been shown to synthesise a high degree of realism at the cost of a high computation load concentrated in solving the scene model [Section 2.4]. A more sophisticated solution to the scene model would provide a faster synthesis of the same realism.

3.2. Fine Tuning the Naive Solution to the Scene Model

Various opportunities arise for fine tuning the naive method. These improvements can reduce average rendering times by an appreciable factor, but do not produce the desired quantum leap in orders of magnitude since they offer no major innovation to the algorithm.

3.2.1. The Significant Path Length Interval

The calculation of a ray's intersection with a CSG object is known as an *object query.* This finds the path length section over which the ray intersects an object. Intersections with the object's surface boundary correspond to the boundary of this section. It is often the case when querying an object that the only surface intersections of interest are those within some *significant path length interval.* Any valid surface intersection must always be after the ray source at some path length greater than zero. A view ray's solution to the scene model finds the nearest surface struck. At any stage of this solution only those surfaces struck before the nearest found to date are significant. The significant interval contracts during a view ray's solution of the scene model. When solving the scene model for an illumination ray to a finitely distant light source, only surfaces struck before reaching this light are significant. The significant interval remains constant during an illumination ray's solution of the scene model. This significant interval can be exploited in various ways to reduce unit object query cost.

3.2.2. Boolean Laws

Each object is modelled by a CSG binary tree, wherein the internal nodes represent binary boolean operations and the leaf nodes primitive solids [Fig 2.2.1a]. The path length section for the 3D construct corresponding to an internal node is found recursively as the

equivalent 1D construct from the sections of the two branches. This incurs only low cost in the trivial path-length sort of the branch surface intersection points. The path length section for the primitive corresponding to a leaf node is found at high floating point arithmetic cost in polynomial construction and solution [APPENDIX C]. The cost of a CSG object query is concentrated at the leaf nodes, which number one more than the internal nodes. A naive complete traversal of the binary tree would visit all leaf nodes incurring high unit query cost, but is generally unnecessary.

Boolean laws may be applied to reduce unit object query cost. Consider an internal intersection construct node. By definition, the node construct is a subset of both branch constructs. Similarly, the path length section for the node construct is a subset of both those for the branch constructs. Whenever the ray misses either branch, their intersection must also be missed. If the first branch query yields an empty path length section, the section for the whole node construct is also known to be empty *without* incurring the tariff of a second query on the other branch.

Suppose the first branch query yields a path length section which is non-empty yet entirely beyond the significant interval. By the same arguments, any surface of the intersected construct can also only be struck beyond the significant interval. The traversal of the node's sub-tree may therefore still terminate without querying the second branch.

Similar savings may be made for other boolean operations [Fig 3.2.2a]. In a realistic scene with many small objects adding fine detail, the majority of objects and their internal constructs will be missed by a given ray to yield an empty parameter section when queried. This scheme may be widely used in such circumstances to limit the number of primitives queried. However, the naive scene model solution will still query each object. These savings can only be made at any internal node after considering the first sub-tree, within which several primitives may yet be queried. Other schemes are required to reduce unit object query cost further.

Fig 3.2.2a: Shortcuts in the Calculation of Path Length Section for a CSG Construct

Notation

Symbol	Meaning
L, R	Left and right branches of CSG construct corresponding to the boolean operation's arguments
S_X	Path length section over which ray is inside construct X
$F(S_X)$	First surface strike of S_X in path length order
μ	Upper limit of significant path length interval

After querying the only left branch, the following shortcuts may be made in the calculation of path length section under the given conditions:

If $S_L=\phi$ or $F(S_L)>\mu$, then

$S_{L \text{ union } R} = S_R$

$S_{L \text{ intersect } R} = \phi$

$S_{L \text{ subtract } R} = \phi$

$S_{L \text{ difference } R} = S_{(L \text{ union } R) \text{ subtract } (L \text{ intersect } R)} = S_{L \text{ union } R} = S_R$

3.2.3. Bounding Volumes

The cost of an object query is dominated by its *geometry* rather than *size* and hence its chance of actually being struck. The query of a sphere involves the construction and solution of a quadratic no matter whether its radius is large or small, even though the latter case is less likely to yield any real roots. To reduce the cost of an object query, any CSG object may be assigned a *bounding volume hierarchy* [Rubin,Whitted;1980: Kay,Kajiya;1986: Goldsmith,Salmon;1987]. Each node in the object's CSG tree is surrounded by a tightly fitting geometrically simple bound at a preprocessing stage. On reaching any CSG node during an object query, the simpler bound is considered before undertaking a more expensive query of the modelled construct, including the root object. In the likely event of this bound being missed the contained construct must be missed too, and the more expensive query is avoided. Otherwise, the recursive object query must continue. The bound query overhead then imposes a greater cost than otherwise at this node. The cost of this overhead is limited however by the geometric simplicity of the bound. Moreover, the scheme may yet yield savings at deeper stages of the query, and if bounds are made sufficiently tight a ray strike should prove the exception rather than the rule.

Once more, a stronger affirmation can be made when the significant interval is bounded above. The path length to the entry point into a struck bound provides a lower limit on that to any surface intersection with the contained construct. If the bound is struck but only beyond the significant interval, the contained construct cannot provide a significant surface intersection [Kay,Kajiya;1986]. The query may then terminate without recursing down the subtree, avoiding high leaf cost.

A typical bound is the box aligned with the world axes or other intersection of *extent slabs* with given normals [Kay,Kajiya;1986]. Clipping planes are placed around a primitive instance at the extent extremes in each normal direction to provide an extent slab of tight fit. These extremes may be located with LaGrangian multipliers [Kay,Kajiya;1986]. Bounds may then be fitted recursively around boolean constructs with the appropriate clipping

planes from their branches' bounds. The query of each plane involves a simple linear polynomial, and the pair forming each extent slab share their linear term. Since such bounds are intersections of these slabs, the previous scheme may avoid querying each slab [Section 3.2.2]. When dealing with the box bound for example, a ray may miss the infinite girder constituting the intersection of the X and Y extent slabs; the Z slab need not then be queried.

The sphere has also been proposed as a bound [Whitted;1980], but suffers two drawbacks. A test for a ray intersecting a sphere can be made in the general case with two dot products [Fig 3.2.3a], involving up to seven floating point multiplications compared to six for an exhaustive world aligned box query. However, this will not yield the path length to any intersection with the sphere bound, and so as a first drawback the method described for ignoring surfaces beyond the significant interval cannot be exploited. The calculation of the path length to an intersection with a sphere requires the solution of a quadratic [APPENDIX C]. This would not be too inconvenient if bounding spheres were generally missed, but this is not the case. Whilst the sphere provides a good fit to constructs of similar extent in each direction, a poor fit is obtained to a construct extending more in some directions than others, such as a long cylinder. The fit of a construct's sphere bound found recursively from those of the branches also tends to be worse than with clipping planes. This presents the second drawback of the spherical bound's poor fit to general CSG objects.

3.2.4. Sturm's Root Test

The query of a primitive isolates the roots of its associated height function, specified as the maximum taken over a number of polynomials [APPENDIX C]. Whilst the analytic isolation of a linear polynomial's root is simple enough, the solution of a quadratic requires somewhat more involved arithmetic. The solution of higher degree polynomials requires extended arithmetic. The query of a torus necessitates the solution of a quartic. Ferrari's method factorises a quartic into two quadratics with a root to a *resolvent cubic* [Korn,Korn;1968a]. The cubic root is found by Cardan's method, which itself involves much arithmetic [Korn,Korn;1968a]. In the *irreducible* case when a cubic has three real roots,

Fig 3.2.3a: Ray Intersection Test for Sphere

Notation

Symbol	Meaning
$\underline{s}$	Ray source relative to sphere centre
$\underline{d}$	Ray direction
R	Sphere Radius

Ray Intersection test

if ($\underline{s}.\underline{s} < R^2$) then — *ray source inside sphere*

    SPHERE HIT

else

if ($\underline{s}.\underline{d} < 0$) then — *sphere behind ray source*

    SPHERE MISSED

else

if ($\underline{s}.\underline{s} - \underline{s}.\underline{d}^2 < R^2$) then — *ray approaches within distance R of sphere centre*

    SPHERE HIT

else ($\underline{s}.\underline{s} - \underline{s}.\underline{d}^2 \geq R^2$) then — *ray always beyond distance R of sphere centre*

    SPHERE MISSED

Cardan's solution cannot be performed within real numbers, but extends to the complex field. A trigonometric solution is then employed [Korn,Korn;1968a]. Unfortunately this proves unstable in practice. The error in the resolvent cubic solution is magnified unacceptably in the quartic solution. To avoid such accumulated numerical errors, the trigonometric solution is tidied up by substitution as the initial root estimate into Newton's iterative method, requiring further extended arithmetic. Polynomials of any degree may be solved with such iterative numerical techniques which all involve extended arithmetic.

The high effort expended in such polynomial solutions is wasted when all roots are isolated outside the significant interval. The existence of roots *within* the significant interval may be checked by *Sturm's method* [Korn,Korn;1968b] before undertaking this such lengthy arithmetic [Fig 3.2.4a]. This comparatively inexpensive existence test may be exploited in a repeated bisection of an interval to reliably locate all roots of a polynomial therein to arbitrary numerical precision. The polynomial may be of any degree.

3.2.5. Reducing the Average Cost of Scene Model Solution for Illumination Rays

The majority of view rays will strike some visible surface in the synthesis of a busy or indoor scene. Each strike spawns an illumination ray to be recursively traced to each light source. If the view model contains multiple light sources the illumination rays traced will then outnumber the view rays. A higher total load is then imposed in synthesising shadows than views by exhaustive search. This search should clearly only be used as a last resort in shadowing.

3.2.6. Self Shadowing Surfaces on Opaque Objects

Whilst transparency is accurately synthesised by ray tracing, this optical property tends to be the exception rather than the rule.

A visible surface point will usually be on an opaque object. It will be shadowed by this object from any light source which is *below* the surface in the sense that a vector towards the light yields a negative dot product with the surface normal. This dot product is already calculated in the synthesis of diffuse radiance [APPENDIX G]. The sign may be checked

Fig 3.2.4a: Sturm's Method for Counting the Roots of a Univariate Polynomial in an Interval

Notation

Symbol	Meaning
$P(x)=\sum_{i=0}^{n} p_i x^i$	Univariate polynomial in x of degree n
$P'(x)$	Derivative of polynomial P(x)
$S^j(x)$	Sturm polynomial of degree j.
mod(P(x),Q(x))	The polynomial remainder after formally dividing the polynomial Q(x) into P(x)
$\overleftrightarrow{X} = [X_I,X_S]$	A univariate interval of specified extremes
$N([y_i]_{i=0}^{n})$	Number of sign changes through the indexed list $[y_i]_{i=0}^{n}$ ignoring zeroes
$R(P(x),\overleftrightarrow{X})$	Root count of polynomial P(x) in the interval $\overleftrightarrow{X}$ where the interval extremes are *not* roots

Sturm's Method

Sturm's method counts the number of roots to a univariate polynomial in an interval, provided that polynomial has no multiple roots [Korn,Korn;1968b]. A sequence of 'n+1' *Sturm polynomials* $(S^j(x))_{j=0}^{n}$ is derived for a polynomial P(x) of degree 'n'. The sequence is similar to that in Euclid's algorithm for the greatest common divisor of two numbers, and is defined by a recurrence relation from a seed pair:

$$S^n(x) = P(x)$$

$$S^{n-1}(x) = P'(x)$$

. . . . . .

$$S^j(x) = -\text{mod}(S^{j+2}(x),S^{j+1}(x)) \text{ for } j \in [0,n-2]$$

. . . . . .

$$S^0(x) = -\text{mod}(S^2(x),S^1(x))$$

Sturm's method counts the roots as follows:

$$R(P(x),\overleftrightarrow{X}) = N([S^j(X_I)]_{j=0}^{n}) - N([S^j(X_S)]_{j=0}^{n})$$

The method still applies when any polynomial from the sequence is scaled by a positive factor, since this has no effect on sign. The recurrence relation may therefore be taken as the alternative

$$S^j(x) = -b_{j+1}^2 \bmod(A(x),B(x)) \text{ for } j \in [0,n-2]$$

where

$$A(x) = S^{j+2}(x);\ B(x) = S^{j+1}(x);\ b_{j+1}^2 \text{ is the leading coefficient of } B(X)$$

As will be seen, this scaling removes all division from the sequence's generation. Sturm's method still applies when the coefficient b_{j+1}^2 is zero.

Any multiple root in a given interval is only counted once. This situation is characterised by the occurrence of a Sturm polynomial $S^j(X)$ of true degree less than 'j' whose x^j coefficient is zero.

Implementation

The manipulation of a Sturm sequence requires some data structure for the representation of a polynomial. A dense representation is particularly convenient whereby a polynomial is stored as its degree and a vector listing its coefficients by increasing order. For example, the representation of the polynomial $P(x) = \sum_{i=0}^{n} p_i x^i$ has the coefficient list $(p_i)_{i=0}^{n}$. This representation is particularly suitable for a polynomial's evaluation at some point α by Horners rule:

$$P(\alpha) = \sum_{i=0}^{n} p_i \alpha^i = p_0 + \alpha\,(\,p_1 + \alpha\,(\,p_2 + \alpha\,(\ldots.\, p_{n-1} + \alpha(p_n))))$$

This may be calculated in an iterative loop incurring 'n' multiplications and 'n' additions. The derivative of a polynomial is also easily found in this format:

$$P'(X) = \sum_{i=0}^{n-1} [i+1]p_{i+1}x^i \text{ with coefficient list } ([i+1]p_{i+1})_{i=0}^{n-1}$$

The two seed polynomials of a Sturm sequence are the given polynomial $S^n(x) = P(x)$ and its derivative $S^{n-1}(x) = P'(x)$. Subsequent polynomials are found according to the given recurrence relation. This finds the polynomial remainder from the quotient of the previous two Sturm polynomials. The quotient's numerator is greater by one in degree than the divisor, and the remainder one less. The derived Sturm polynomial is this remainder scaled by the negated square of the divisor's leading coefficient:

Such remainders are easily found in the coefficient list format. Consider finding the polynomial remainder R(x) = mod(A(x),B(x)) for the Sturm polynomials A(x), B(x) defined as above by formal long division:

$$\begin{array}{r|l} & \dfrac{a_{j+2}}{b_{j+1}}x + \dfrac{a_{j+1}b_{j+1}-a_{j+2}b_j}{b_{j+1}^2} \\ \sum_{i=0}^{j+1} b_i x^i & \sum_{i=0}^{j+2} a_i x^i \\ & -\sum_{i=0}^{j+2} \dfrac{a_{j+2}}{b_{j+1}} b_{i-1} x^i \\ \hline & = \sum_{i=0}^{j+1} \left[\dfrac{a_i b_{j+1} - a_{j+2} b_{i-1}}{b_{j+1}} \right] x^i \\ & -\sum_{i=0}^{j+1} \left[\dfrac{a_{j+1}b_{j+1}-a_{j+2}b_j}{b_{j+1}^2} \right] b_i x^i \\ \hline & = \sum_{i=0}^{j} \left[\dfrac{(a_i b_{j+1} - a_{j+2} b_{i-1}) b_{j+1} - (a_{j+1}b_{j+1}-a_{j+2}b_j) b_i}{b_{j+1}^2} \right] x^i \end{array}$$

Therefore

$$R(x) = \sum_{i=0}^{j} \left[\frac{Ua_i - Vb_{i-1} - Wb_i}{b_{j+1}^2} \right] x^i$$

with the constants

$$U = b_{j+1}^2;\ V = a_{j+2}b_{j+1};\ W = a_{j+1}b_{j+1} - a_{j+2}b_j;\ b_{-1} = 0$$

and so

$$S^j(x) = -b_{j+1}^2 R(x) = \sum_{i=0}^{j} (-Ua_i + Vb_{i-1} + Wb_i) x^i$$

with coefficient list $(-Ua_i + Vb_{i-1} + Wb_i)_{i=0}^{j}$

The coefficient lists of successive Sturm polynomials are found by this relation. A triangular matrix of these coefficients for the entire sequence is found by iteration.

Sturm's method may be used to check for existence of roots to a given polynomial within the significant path length interval. This interval has a minimum of zero, at which point each polynomial in the sequence simply evaluates to its constant term. The number of sign changes

through the evaluated polynomial list is then simply the count through the list of constants. If this count proves to be zero there are no roots in the significant interval. Otherwise, the number of sign changes is found through the list of polynomials evaluated by Horner's rule at the interval's maximum. The number of roots in the interval is then the difference in these sign change counts.

Sturm's method may be adapted to locate the roots of a polynomial within a given interval to arbitrary numerical precision with a binary chop, rather than merely testing for their existence. The interval is recursively bisected until the current interval is either found to contain no roots or is of smaller width than the given precision. Any roots are conveniently found in increasing order by considering the halves of each bisected interval in that order. The convergence of the binary chop may be slower than other techniques such as Newton's iteration. However, this method is far more robust and is sufficiently reliable to *always* locate *all* polynomial roots when ray tracing.

The compound coefficients of successively generated Sturm polynomials may grow rapidly with the described iteration. When starting from a polynomial of high degree these coefficients may explode in modulus causing numerical overflow. This is easily avoided by normalising each successive polynomial to unit infinity norm over its coefficient vector. Scaling by a positive factor does not effect the sign counts and prevents such overflow. Under 'usual' circumstances this normalisation is only necessary for polynomials of 'high' degree, typically six or more.

before recursively tracing an illumination ray from an opaque object surface, often avoiding the exhaustive search at no extra cost.

3.2.7. Spatial Shadow Coherency In Images

When synthesising an image in raster order, successive visible points found as view ray solutions to the scene model will usually be for primary view rays through adjacent pixels in the view screen. They will tend to be on the same surface, under similar lighting conditions. This coherency may be exploited by assigning each light source a record of the most recent opaque object found to cast a full shadow when solving the scene model for illumination rays [Haines,Greenberg;1986]. The record is initialised to an arbitrary object and is updated each time a new object is found to cast a full shadow from the light on a visible surface. The record is examined before recursively tracing an illumination ray to the light source. A single query is made of the corresponding opaque object and if this casts a full shadow an exhaustive object query search is again avoided. Otherwise, the search is undertaken without a repeated query for this object, so that the check involves no significant extra cost. The record is only updated during such a search.

3.3. The Effectiveness of Fine Tuning the Naive Solution to the Scene Model

The techniques for reducing unit object query cost do provide savings in the naive solution of the scene model for a view ray. However each object is still queried in an exhaustive search. Whilst these savings are worthwhile, the computational load of solution is still linear in object count which may be high in realistically detailed scenes.

The techniques for fine tuning the solution for an illumination ray all provide an opportunity to avoid an exhaustive search whilst imposing only low overhead themselves. They are certainly also worthwhile, but their impact is limited since in the general case the solution of the scene model returns to the catch-all exhaustive search. Even if these techniques succeed in filtering out half of these default cases, thereby halving the significant portion of the computational load, synthesis times would only improve by a factor of two.

An improvement in orders of magnitude clearly requires a more efficient default solution than naive exhaustive search. This must be avoided always rather than merely sometimes if viable rendering times are to be achieved.

3.4. Fast and General Solutions to the Scene Model

Much recent research has addressed this major obstacle in ray tracing. Some attempts have been made to reduce synthesis times by computing at a faster rate rather than reducing the computational load. Various researchers have attempted to harness the more computationally powerful hardware becoming available such as multi-processor and transputer systems [Dippe,Swensen;1984: Muller;1987: Plunkett,Bailer;1985: Williams,Buxton,Buxton;1986]. Whilst hardware solutions do indeed pay dividends they do not preclude gains offered by more sophisticated software solutions to the scene model. Both avenues are worthy of exploration. Due to limitations in the hardware available for the research culminating in this thesis, only software solutions on a single processor system have been considered. These are applicable to a range of computing hardware. They do not rely on multi-processor systems, which though becoming more readily available are still less common than single processor architectures.

3.4.1. Avoiding Exhaustive Search

The global scene model may contain many objects. Only a few of these will be significant to a ray's scene model solution. Indeed only one is significant for a view ray - that with the nearest intersected surface. Many other objects are 'obviously' insignificant. Consider a scene model of a room with a Venetian blind across a window. The blind comprises many slats, each modelled by a stretched cube. Suppose the view model specifies a viewer looking into the centre of the room with the window behind him. None of the slats are significant to any primary view ray; the intersection of any such ray with a slat would be at a negative path length, outside the significant interval. The entire blind should be 'obviously' rejected from any local view ray solution. It should not be rejected from the global scene model however, since the slats may cast shadows within the room. These will

be synthesised by tracing illumination rays, to which the blind is significant. An efficient scene model solution should identify and reject insignificant objects in the global scene for a given ray.

Various algorithms have been proposed as efficient general solutions to the scene model for view and illumination rays [Amanatides;1984: Avro,Kirk;1987: Fujimoto et al;1986: Glassner;1984,1988: Goldsmith,Salmon;1987: Haines,Greenberg;1986: Heckbert,Hanrahan;1984: Kaplan;1985: Kay,Kajiya;1986: Marsh;1987: Wyvill et al;1986]. These exploit coherency to simplify the problem with a *divide and conquer* approach. They identify and reject objects which are locally insignificant to the scene model solution, thereby avoiding an exhaustive search. This is often carried out at a preprocessing stage but may be performed dynamically during image synthesis. Two broad camps may be identified - algorithms exploiting ray coherency in the view model and algorithms exploiting spatial coherency in the scene model.

3.4.2. Solutions Exploiting Ray Coherency in the View Model

These methods are motivated by the observation that similar rays generally have similar intersections with scene objects. In particular, they will tend to miss the same objects and intersect the others in the same path length order. Moreover, many rays generated by the view model are indeed similar and may be partitioned into a few equivalence classes. The concept of 'similar rays' varies slightly between proposed algorithms, but broadly applies to rays of close or often equal source position and close spherical direction. Since all ray direction vectors are of unit length, they occupy a unit sphere surface parameterised by 2D latitude and longitude rather than full 3D space. Methods insisting that similar rays have identical sources are often generalised from 2D acceleration techniques in previous image synthesis algorithms such as Warnock's algorithm [Haines,Greenberg;1986: Heckbert,Hanrahan;1984: Amanatides;1984]. All primary view rays share a common source at the view point and are traced in successively similar directions when synthesising in raster order. Since an illumination ray's solution to the scene model may consider intersected objects in any path-length order, all illumination rays associated with a finitely

displaced light source may be traced from, rather than towards, that light and thereby also share a common source.

The positions of the viewer and finitely displaced light sources are key points in the scene being common *focal* points for many rays. They are known from the view model before the pixel by pixel synthesis. Primary view rays and illumination rays may therefore be divided into a partition of groups each containing rays with the same source and similar directions [Haines,Greenberg;1986]. For primary view rays each group usually corresponds to those rays passing through a given tile of the image. The silhouette of every object is projected down onto the screen to find those yielding non-empty intersections within each group. This provides a reduced list of objects significant to each group. When this involves difficult geometry, a simpler bound may be projected instead [Haines,Greenberg;1986: Avro,Kirk;1987]. For illumination rays, each finitely displaced light source may be surrounded by a cube of six such screens with given pixelation [Haines,Greenberg;1986]. The significant object lists for the groups on each screen are found similarly. For an infinitely displaced light source, a single pixelated screen large enough to shadow the entire scene from the light may be orientated normal to the direction of illumination. These group lists reject objects perpendicularly distant from the path of their associated rays. As described however this ray division by 2D direction loses path length depth information for each significant object and so cannot avoid querying objects perpendicularly near the ray path yet outside the significant interval. Depth information can be partially retained by allocating each object in a group a record of the path length extremes of intersection with the rays in that group [Haines,Greenberg;1986]. When this involves difficult geometry, estimates may be obtained using simpler bounds on objects and spherical ray direction. The group may be priority sorted according to these extremes to facilitate the rejection of insignificant surfaces. A bit record may also be allocated indicating whether the object surface is struck by all rays in the group or only some. This provides further savings for illumination rays when an opaque object casts a full shadow.

The sources of secondary and subsequent generations of view rays are not so spatially coherent, but distributed between many surface points. Moreover, these surface points are not known before synthesis. A 2D grouping on ray direction cannot be preprocessed without knowing these points for silhouette projection. Some algorithms have been proposed which dynamically group rays by 2D direction during synthesis to allow for arbitrary ray source [Heckbert,Hanrahan;1984: Amanatides;1984]. A more general ray grouping has been proposed not only on 2D direction but also 3D source position yielding 5D hypercube equivalence classes [Avro,Kirk;1987]. Rays need no longer have the same source to be classified as similar, but only close sources within some rectangle 3D box space grouping. These hypercube groups are created dynamically during image synthesis as secondary generations of rays are traced from new scene regions. The inherent complex geometry for a general scene model necessitates geometrically simpler bounding volume approximations.

3.4.3. Solutions Exploiting Object Coherency in the Scene Model

These methods are motivated by the observation that close points constituting small scene regions generally tend to be on surfaces of similar objects, if any. The number of objects with surfaces passing through such a local scene region is generally far less than over the global scene. These are the only objects significant to the scene model solution within the region. Any object which is entirely inside or outside a scene region is said to be *homogeneous* with respect to the region. Otherwise part of the object's surface is within the voxel and this is said to be *heterogeneous* with respect to the region. An object is significant within a region when it is heterogeneous. An exhaustive object search may be avoided by considering only the heterogeneous objects of those regions visited by a ray. This avoids querying objects perpendicularly distant from the ray path. Moreover, by considering these regions in path length order all those outside the significant interval are easily ignored together with their associated objects. This avoids querying objects perpendicularly close to the ray path but distant from the ray source. The most common region is the rectangular box aligned with the world axes [Fujimoto et al;1986:

Glassner;1984,1988: Goldsmith,Salmon;1987: Whitted;1980: Wyvill et al;1986] although other convex polyhedra have also been used [Kay,Kajiya;1986].

Recent research has addressed two major types of scene decomposition into hierarchies of such regions.

A bounding volume hierarchy may be built bottom-up from those of the scene objects [Goldsmith,Salmon;1987: Kay,Kajiya;1986: Rubin,Whitted;1980]. This hierarchy is a generalisation of that already described within CSG objects [Section 3.2.3]. Whilst the latter is an *intra* object hierarchy within each object, the former is an *inter* object hierarchy within the scene in which each object forms a leaf rather than the root. The scene is thereby modelled as a single entity rather like the union of all objects, but each leaf now has its own constituent material. By definition, the leaf regions of this hierarchy are the object bounds. These are generally dispersed irregularly throughout the scene without forming a partition thereof. Some scene points may not be included in any leaf region, whilst others may be in several overlapping object bounds. On one hand, this scene hierarchy reduces the computational load of ray tracing in the same manner as CSG construct bound hierarchies [Section 3.2.3]. On the other, the hierarchies' constructions differ greatly. The CSG hierarchy inherits structure from the binary CSG description tree, and construction is straightforward. However, there is generally no such obvious choice for the inter object scene hierarchy. The scene model may have been constructed with some hierarchy but this will have been designed to facilitate modelling rather than image synthesis. The choice of possible hierarchies is enormous for any non-trivial object count. Hierarchies need not be limited to binary forms, or indeed any constant branching ratio. This extensive choice results in a wide range of potential savings in computation. The construction of an efficient hierarchy offering a good trade in the costs of image synthesis for construction is non-trivial.

Alternatively, a scene partition hierarchy may be built top-down from a region containing the entire scene [Fujimoto et al;1986: Glassner;1984,1988: Kaplan;1985: Marsh;1987: Wyvill et al;1986]. Such hierarchies are well established in 3D modelling applications [Wyvill et

al;1986]. The scene may be decomposed with an octtree which recursively splits regions by simultaneous bisection in each dimension [Fujimoto et at;1986: Glassner;1984,1988: Wyvill et al;1986]. The bin-tree has been proposed, and bisects in a single cycled dimension [Kaplan;1985]. The regular grid partition or *Spatial Enumeration* has also been employed, comprising cells of uniform size [Fujimoto et al;1986: Marsh;1987]. These methods generate regular local scene regions. Each is called a *voxel* volume element as a 3D generalisation of the 2D *pixel* picture element. By definition, leaf voxels in such schemes partition scene space. Each scene point is in one and only one voxel.

The construction of these top-down hierarchies is straightforward. Voxels are recursively split until either of two termination criteria is satisfied. The *simplicity* criterion tests for a voxel of heterogeneous object count below some upper limit. Any such voxel is considered sufficiently simple and the decomposition has been successful in this case. The *depth* criterion tests for a voxel of maximum permitted depth within the decomposition and prevents runaway. Any such voxel is considered too complex to be made simple, and decomposition has been at best a partial success in this case. However the voxel will be comparatively small being at this maximum depth and therefore rarely navigated by a ray. Since such hierarchies are both formed and queried top-down, they may be built during image synthesis by *lazy construction.*

3.5. The Utility of General Solutions to the Scene Model

Each group of acceleration techniques share common characteristics.

3.5.1. The Utility of Solutions Exploiting Ray Coherency

Whilst methods preprocessing rays of common source into groups of similar 2D direction offer a faster solution to the scene model where applicable, they are not fully general as they do not allow for secondary view rays [Haines,Greenberg;1986]. Moreover, some methods severely restrict the scene model to make the complex geometry involved in identifying significant objects more tractable, typically allowing only polyhedra or spheres without CSG [Heckbert,Hanrahan;1984: Amanatides;1984]. This would not be an

overwhelming drawback when synthesising only the limited realism already produced by other schemes. Complex geometries in a scene model may be approximated with polyhedra and smoothly shaded with Gouraud or Phong techniques [Marsh;1987]. Reflected view may be approximated to one level with an environment map [Amanatides;1987] and refracted view with similar texture maps. However, the motivation for ray tracing is the high degree of realism synthesised with accurate solutions to general view and scene models. Approximations in shading and visible surface calculations can forfeit potential realism and so 'throw the baby out with the bath water'. Faster, well established scan line algorithms are more appropriate for the synthesis of limited realism. A truly realistic image of a general indoor scene must accurately synthesise object shape, shadows and several generations of views. Whilst perfect mirrors are rare, the radiance from many objects will contain some reflected view component such as polished floors, glass windows, plastic and metal surfaces and even gloss painted walls. Perfectly transparent objects are also uncommon, but any radiance from a transparent body such as a glass object, liquid or volume density will generally contain some refracted view component with refraction though a non-trivial index. These details are not unimportant minutiae, but must be realistically synthesised if the goal of photo-realism is ever to be achieved. This requires an efficient solution of the general scene model whose application is *completely* unrestricted.

The decomposition of rays into 5D hypercubes attempts to overcome these problems [Avro,Kirk;1987]. This supports a general view model, dealing with rays of arbitrary source for application to secondary view rays. However, the geometry required to directly identify objects significant to the solution of a general scene model within a hypercube region would be extremely complex. Both scene objects and hypercube projections are approximated with bounding volumes to ease the problem. Convex polyhedral object bounds are used with pyramidal hypercube projections, and spherical object bounds with conical hypercube projection bounds. The degree of approximation necessary to simplify the problem to a feasible level can be great, resulting in many objects being wrongly identified as significant

to a hypercube. This is particularly true of those objects yielding a poor bound approximation such as a non-convex torus or CSG object. The holes in such objects may be wrongly identified as projecting a surface within a hypercube. Bounding approximations can also result in poor estimates of an object's path length depth extremes within a group of rays.

Schemes relying on a view model decomposition share a common drawback when synthesising an animated sequence of images. The majority of a scene often consists of static or background objects such as the furniture of an indoor scene. Often only a few objects in the scene model change between frames, such as a dynamic figure walking through a static room. However, the view model often changes radically when dealing with a moving viewer such as in a fly past or walk through simulation. This generally low temporal coherency of the *view* model necessitates a repeated decomposition from scratch for each image. However, the generally high temporal coherency of the *scene* model may be exploited by schemes decomposing this model. They need only reprocess the scene decomposition for the limited number of dynamic objects over successive frames. A similar situation can occur even when synthesising a single static image. Scene models are easily visualised and a correct model is often produced first time, especially by automated computer modellers. However, several instances of the view model may have to be tried before achieving the desired perspective, visible surfaces, surface shading and other optical effects. Each instance requires a new decomposition. This observation is somewhat subjective, but is often the case in practice.

3.5.2. The Utility of Solutions Exploiting Object Coherency

Scene decompositions offer fully general solutions to the scene model. The bounding volume hierarchy method is particularly easy to implement if already used within CSG constructs, as much of the necessary code is already available. The hierarchy for an animated scene may be divided at the root level into a small branch of dynamic objects and a larger one of static objects. Only the former need be reprocessed for each synthesised frame.

The use and automatic generation of bounding volume hierarchies is discussed in chapter four. However, simple convex bounding volumes can once more lead to over-approximation. A ray along the rotational axis of a torus through the central hole will miss this primitive. The convex hull is clearly struck however. Since this is the intersection of all convex bounds, any convex bound must also be struck. The torus would be wrongly identified as significant to the scene model solution and subsequently queried. The common rectangular box bound aligned with the world axes gives a poor fit to a long object not aligned with any world axis. The former may often be struck when the latter is missed, resulting in similar problems. Attempts to introduce other bounds of better fit can be counter-productive due to increased geometric complexity. Clearly the bound with the best fit to any object is the object itself, but this offers no savings to an object query.

Partition hierarchies overcome this drawback by decomposing right down to an object's surface rather than merely to the body. Any convex bounding volume must contain the entire negative region of its content's height function. Bounding volume hierarchies are built bottom-up, and so have an inherent fixed depth. Leaves must contain entire objects, or primitives for the CSG hierarchy case. Partition hierarchies are built top-down however, and can recursively decompose to any level. By decomposing to a great depth, leaves are made to focus not merely on object bodies but on surfaces. They then contain only the salient roots of height functions rather than being blocked at the less significant negative regions. Partition hierarchies can break through the barrier extremes of these negative regions to home in on any internal roots, and so have a greater impact in reducing rendering times. Alternatively, construction times may be reduced by decomposing to a lesser depth. A partition hierarchy may be constructed dynamically as required during synthesis as observed previously [Section 3.4.3], whilst a bounding volume hierarchy must be entirely preprocessed. The partition hierarchy is clearly more flexible.

The use and automatic generation of the regular grid partition is discussed in chapter five, whilst chapter six addresses the octtree decomposition.

Chapter 4: Bounding Volume Hierarchies

Synopsis:

Chapter four addresses the decomposition of a scene by a bounding volume hierarchy. A single query of a branch bound within such a hierarchy can avoid querying all the object descendants. An efficiency metric is derived for bounding volume hierarchies. This is used to develop an algorithm for the generation of optimal or at least quasi-optimal hierarchies.

4.1. The Simplification of the Scene Model with Bounding Volume Hierarchies

A ray is traced through the bound hierarchy [Section 3.4.3] to solve the scene model. The hierarchy is traversed top down, querying each bound node for contents significant to the scene model solution. The total hierarchy query cost should be minimised for fast image synthesis by both node count and unit node cost.

4.1.1. Minimising Query Count

Many researchers have addressed the minimisation of query count which is now well understood [Kay,Kajiya;1986: Rubin,Whitted;1980]. The number of bound nodes which are candidates to contain significant objects is recursively reduced by pruning entire branches from the hierarchal tree. If a ray misses a branch's bound, no contained object can be struck and so the whole branch is rejected from further traversal. If hit however a further test is made for a contained surface possibly being within the significant interval [Section 3.2.3]. The path length to a struck bound is immediately available from the bound query. Clearly, this is a lower limit on the distance to *any* surface intersection with the contained objects. Should this be *beyond* the significant interval, then by transitivity the entire branch may still be pruned. Otherwise, the bound's contents remain significant. If the bound is a leaf, the single contained object is queried. Otherwise, the traversal descends over the branches.

This recursion is usually performed *breadth* rather than *depth* first to maximise the degree of pruning [Kay,Kajiya;1986]. When dealing with a view ray, a high object query cost incurred in the immediate descent down the first branch of the struck bound is wasted if this branch is subsequently found to be beyond an object intersection within another branch. The traversal rather queries all branch bounds, sorts those struck by increasing path length, and then recurses in this order so avoiding unnecessary computation. The scene model solution is complete when this traversal terminates, hopefully having pruned many entire branches.

4.1.2. Minimising the Cost of a Single Bound Query

Research on the minimisation of unit query cost has not been so widely reported. The cost of examining the sign of a real variable is usually negligible compared to that of floating point arithmetic in its computation. This may be exploited to trivially identify many cases where bounds are missed by a ray *without* recourse to arithmetic, much as in the Cohen-Sutherland 2D clipping algorithm [Foley,Van Dam;1984].

For clarity, the 2D case for box bounds aligned with the world axes is described [Fig 4.1.2a]. The method immediately generalises to the 3D case with slab intersection bounds of given normals. This query algorithm is of lower average unit cost than others described for a single bound [Kay,Kajiya;1986: Goldsmith,Salmon;1987].

4.2. Exploiting Clipping Plane Inheritance

Further opportunities arise for computational savings during queries in a hierarchy of bounds. When forming a node's bound from those of its branches, the clipping plane extremes of each slab in the bound are simply those extremes taken over all branch bounds. During the top down traversal of the hierarchy each clipping plane of an internal node's bound will therefore be inherited by one or another of the branch bounds. This may be exploited to avoid unnecessary repetition of clipping plane intersection. The path length to each intersected clipping plane may be forwarded from a struck bound to the branch queries rather than being recalculated.

This has an immediate implication on the preferred branching ratio of a hierarchy. An internal node could have any multiple branch count. These branches are queried independently. However, any pair could be bounded together to be simultaneously rejected with a single query. Provided the pair's bound is sufficiently tight that the floating point arithmetic for this query would have been required by the branch queries anyway, only negligible extra cost is incurred from floating point examination. An inductive argument then shows the optimal branching ratio to be two. Binary hierarchies are therefore most efficient when exploiting clipping plane inheritance to remember rather than recalculate

Fig 4.1.2a: Bound Queries

The bound in the 2D case below is formed by the intersection of two *extent slabs*. These have unit normals $\underline{x} = (1,0)$ and $\underline{y} = (0,1)$. The query method immediately generalises to 3D bounds formed by the intersection of any number of slabs with given normals.

Consider a query of the slab intersection bound by the given ray of source $\underline{s}$ and unit direction $\underline{d}$.

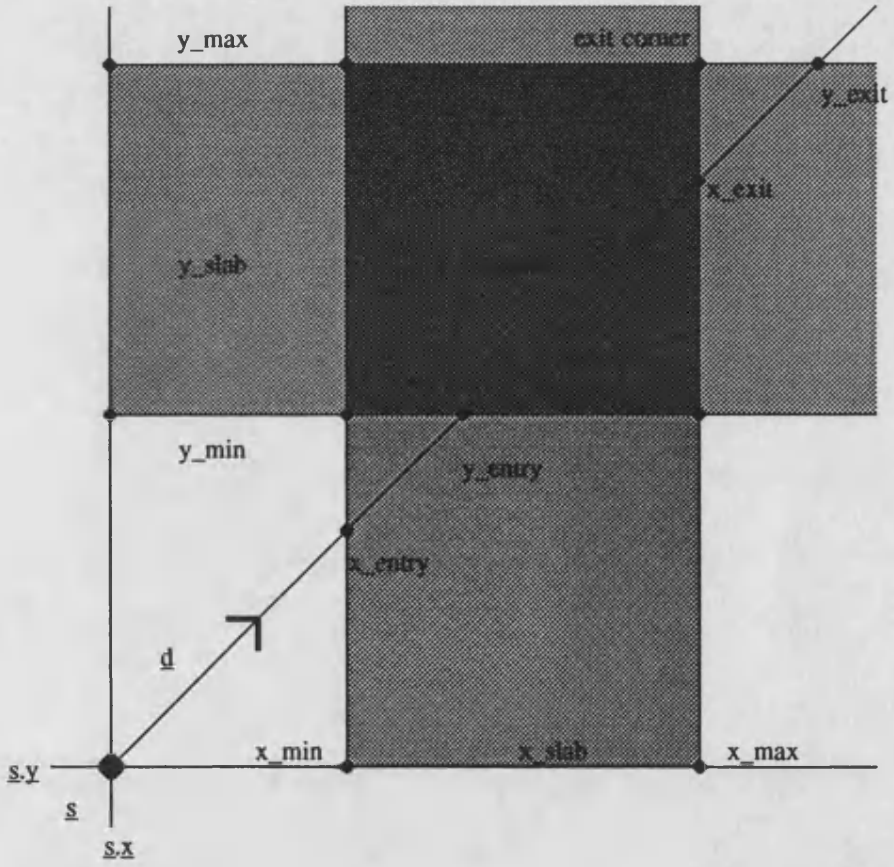


Fig 4.1.2a

Two clipping planes border each extent slab. These are at the slab's minimum and maximum extremes, displaced at 'slab_min' and 'slab_max' respectively in the slab normal direction $\underline{n}$ from the world origin. The projections of the ray's source and direction onto each slab's normal are $\underline{s}.\underline{n}$ and $\underline{d}.\underline{n}$. These are constant for a given ray, and need only be calculated once for all that ray's queries [Kay,Kajiya;1986].

The extremes of each extent slab are displaced at 'slab_min – $\underline{s}.\underline{n}$' and 'slab_max – $\underline{s}.\underline{n}$' from the ray source in the slab normal direction. The ray spans this direction $\underline{n}$ at the rate $\underline{d}.\underline{n}$ with respect to path length. It is therefore inside each slab over the path length section (slab_entry, slab_exit) defined by

$$\left(\frac{\text{slab\_min}-\underline{s}.\underline{n}}{\underline{d}.\underline{n}}, \frac{\text{slab\_max}-\underline{s}.\underline{n}}{\underline{d}.\underline{n}} \right) \text{ for } \underline{d}.\underline{n} > 0$$

$$\left(\frac{\text{slab\_max}-\underline{s}.\underline{n}}{\underline{d}.\underline{n}}, \frac{\text{slab\_min}-\underline{s}.\underline{n}}{\underline{d}.\underline{n}} \right) \text{ for } \underline{d}.\underline{n} < 0$$

$$(-\infty, +\infty) \text{ for } \underline{d}.\underline{n}=0 \;\&\; 0 \in [\text{slab\_min}-\underline{s}.\underline{n}, \text{slab\_max}-\underline{s}.\underline{n}] \text{ (ray source inside slab)}$$

$$(+\infty, +\infty) \text{ for } \underline{d}.\underline{n}=0 \;\&\; 0 \notin [\text{slab\_min}-\underline{s}.\underline{n}, \text{slab\_max}-\underline{s}.\underline{n}] \text{ (ray source outside slab)}$$

The ray intersects the bound over the path length section (bound_entry, bound_exit) defined by

$$(\text{bound\_entry}, \text{bound\_exit}) = (\max_{\text{slab}}\{\text{slab\_entry}\}, \min_{\text{slab}}\{\text{slab\_exit}\})$$

A typical bound query algorithm calculates *all* clipping plane intersections to find this interval [Kay,Kajiya;1986].

However, the ray is actually defined only for non-negative path lengths. This interval is therefore more strictly

$$(\max\{\max_{\text{slab}}\{\text{slab\_entry}\},0\}, \max\{\min_{\text{slab}}\{\text{slab\_exit}\},0\})$$

The ray can only strike the bound if the maximum of this interval is positive. This is only possible if the maximum of each slab's interval is positive, that is

$$\text{sign} (\text{slab\_exit\_extreme} - \underline{s}.\underline{n}) \times \text{sign} (\underline{d}.\underline{n}) > 0$$

More intuitively let an *exit corner* be an intersection of slab exit planes equal in number to the world dimensions 'D'. The ray can only strike the bound if at least one exit corner is in the same D-ant relative to the ray source $\underline{s}$ as the ray direction $\underline{d}$.

Any bound entirely behind the ray source may be rejected by applying this sign test to each slab *without* floating point multiplication. Such cases may be rare for primary view rays from a remote

viewer facing the scene body, but will often occur for secondary rays originating within the scene.

The query need only proceed if this sign condition is satisfied by *every* slab. Floating point multiplication may still be avoided if the ray source proves to be within each slab. The ray is then known to enter the bound at path length zero. This occurs when the minimum of each slab's path length section is negative, that is

$$\text{sign} (\text{slab\_entry\_extreme} - \underline{s}.\underline{n}) \times \text{sign} (\underline{d}.\underline{n}) < 0$$

Floating point multiplication is only needed when at least one slab has an entry plane in front of the ray source, in the sense that this sign condition is not satisfied. Each slab is considered in turn. The path length to the slab entry plane is calculated when in front of the ray source, but simply set to zero otherwise. The minimum of the bound's path length section is updated accordingly. The path length to the slab exit plane is also calculated, and the maximum of the bound's path length section updated similarly. The interval's minimum is tested for exceeding the maximum. If so, the ray is known to miss the bound and the query is complete. This may occur after considering only two slabs and incurring just two floating point multiplications.

The query may complete with no opportunity for such short cuts. However, the path length section over which the ray is inside the bound has been found and may be exploited for other savings. The path length to the bound entry plane provides a lower limit to any intersection with the contents [Section 3.2.3]. The existence of path length roots corresponding to surface intersections within the interval may be tested by Sturm's method [Section 3.2.4].

Fig 4.1.2a

clipping plane intersections during the traversal.

4.3. The Generalised Application of Bounding Volume Hierarchies

So far only queries on inter-object hierarchies have been described. However these schemes may be adapted slightly to apply equally well to intra-object hierarchies within objects.

4.3.1. Bounding Volume Binary Hierarchies in CSG objects

Clipping plane inheritance may also be exploited in the query of a bounding volume hierarchy within a CSG object [Section 3.2.3]. The clipping plane extremes of an extent slab in a node's bound are no longer necessarily the extremes taken over the branch bounds - consider the intersection of two boxes. However, the clipping planes of a node's bound will still be inherited in some way by the branch bounds. An entire CSG branch is still pruned when a ray misses its bound. If hit the appropriate recursion over the branches depends on the associated boolean operation, as only surfaces struck within the significant interval are of interest.

For a union, both branch bounds are queried and those struck are sorted by increasing path length. The traversal recurses through this list whilst the path length to the current branch bound is within the significant interval.

Whenever an intersection node's bound is struck both branch bounds must also be struck by the definition of intersection. The clipping plane intersections of any branch's bound are calculated none the less before recursion for subsequent inheritance. The branch with the more distant bound is recursed down first, and only if this yields a non-empty intersection within the significant interval need the other be considered. The further branch is simply that inheriting the ray's entry plane into the node's bound.

For a subtraction, the left branch is recursed down first. Only if this yields a surface intersection within the significant interval need the other be considered. The left branch bound need not be queried since it inherits all the parent's clipping planes.

The symmetric difference operation may be expanded in terms of the union, subtraction and intersection. The appropriate order of recursion is then seen to be the same as for union.

4.3.2. Bounding Volume Hierarchies within Polyhedral Boundary Representations

The polyhedral approximation is a common model for the boundary representation of complex geometries. This approximates an object's boundary with many polygonal facets, typically triangles. Similar models use fewer but more complex local surface approximations, such as bi-cubic patches [Catmull;1978].

Whilst the CSG model supports many geometries without resorting to approximation, extensive polyhedral modelling systems have been developed for the boundary representation of a wide range of objects. The large number of polyhedral models already available would provide an extensive modelling environment for ray tracing when allied with CSG and smooth shading techniques.

Each polygonal facet of a polyhedron is easy to ray trace. However, the entire path length section of a ray's intersection with a polyhedron is required to fully integrate the polyhedron as a new primitive in the CSG scene model supporting all boolean operations. A naive ray intersection test on each facet followed by a sort into increasing path length is clearly inefficient for models containing thousands of facets. A hierarchy of bounding volumes around the polygonal facets of a polyhedron offers computational savings in the same manner as an inter-object hierarchy.

4.3.3. Three Applications of Bounding Volume Hierarchies

In summary hierarchies may be built from polygonal facets or other patches into a CSG primitive, from CSG primitives into an object, and from objects into a scene [Fig 4.3.3a].

4.4. Data Structures for Hierarchy Representation

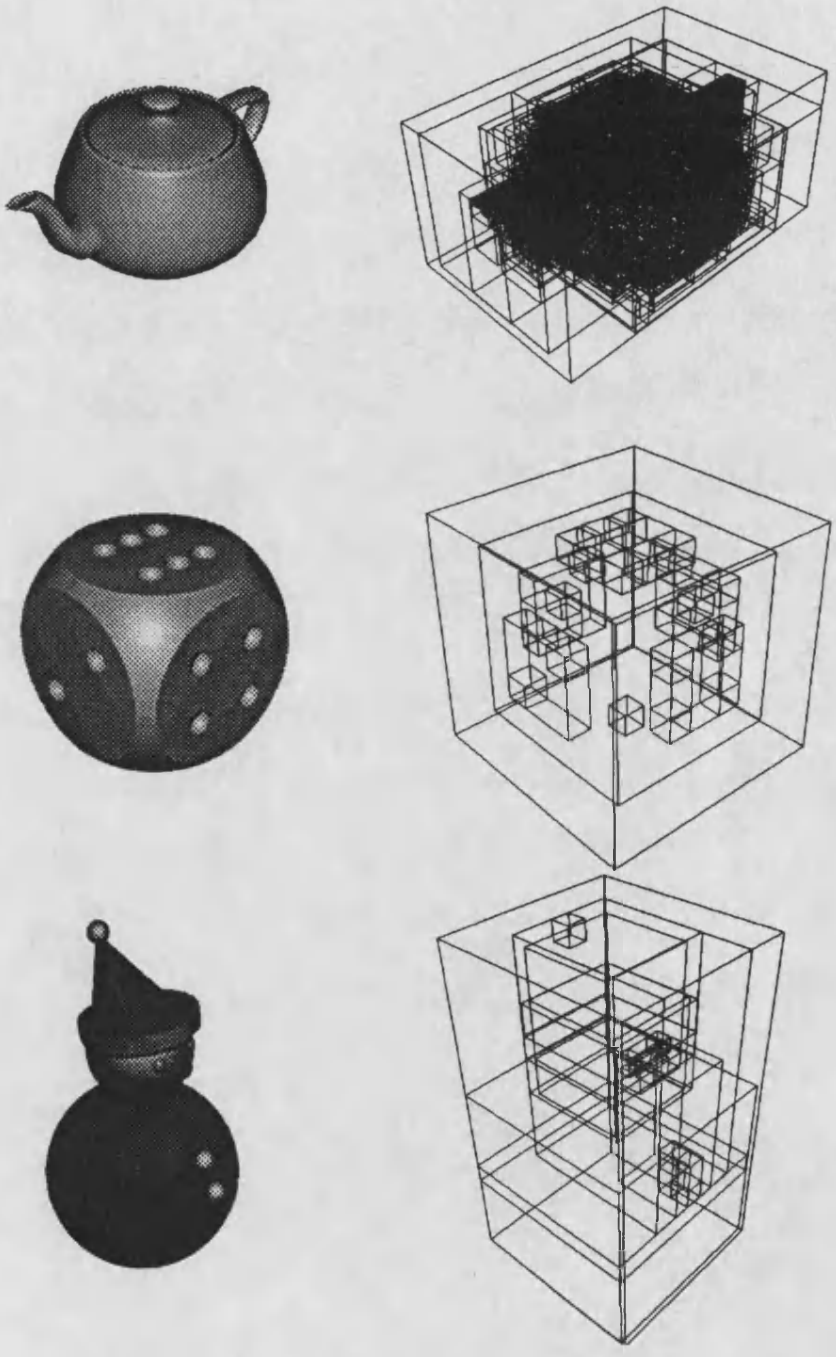
Any binary hierarchy is canonically described by a binary tree. This may be represented as separate preorder lists of the internal and leaf nodes. Given that the leaves must be stored

Fig 4.3.3a: Three Uses of Bounding Volume Hierarchies

Top - built from polygons within a polyhedral boundary approximation

Middle - built from primitives within a CSG object

Bottom - built from objects within a scene



anyway, the only extra memory cost of this representation is for the internal nodes which number one less than the leaves.

Each internal node has

- Two branches - which should be easily locatable from the parent, for efficient tree pruning during traversal;
- An associated slab intersection bound - all of whose clipping planes are inherited by the branch bounds in some way or another;
- For CSG nodes only - a boolean operator.

A tree traversal must be able to recognise any leaf node as such rather than an internal node.

All these data may be represented at an internal node by

- A count of the leaves in the left branch - held as an integer;
- A set of slab clipping plane extremes - in floating point format relative to the world origin;
- A clipping plane inheritance record - for the branches, often held in a single byte;
- For CSG nodes only - a boolean operator byte.

Branch pointers are not required. Any branch's location is immediately apparent in the hierarchy preordering from the leaf count in the left branch. This also flags when a leaf is reached.

The clipping plane extremes held at an internal node are for those *uninherited* by the branch bounds.

The inheritance record defines which branch inherits each clipping plane. It constitutes a 'will' of bit flags, one per clipping plane. Box bounds have six clipping planes and so require only a single byte record. An extra field is used for CSG nodes, as the branches need not *necessarily* be bounded. Whilst the intersection of a concentric sphere and plane is bounded, the plane itself is not. Each branch is allocated one bit flagging whether it is indeed bounded. The inheritance record still fits into a single byte for box bounds.

4.5. The Query of a Bounding Volume Hierarchy

A recursive ray query of a hierarchy of these structures follows a top down traversal, starting at the root bound. If this is not struck, the ray misses every object and the scene model is trivially solved. If the root bound is struck but contains only a single object, this object is queried without incurring further cost within the hierarchy. Otherwise, the root node is internal and the query recurses over the branches. The bound of each internal node reached during the subsequent traversal is always known to be struck and all descendant leaves remain significant to the scene model's solution. The following data are passed down from the parent:

- the path lengths to the clipping plane intersections of the node's bound;
- the leaf count below the node;
- the count of leaves traversed so far - including those pruned.

The branches to be recursed down are determined in an appropriate order [Section 4.3.1]. Queries of branch bounds calculate clipping plane intersections only where not inherited. The unpruned branches are tested for being leaves. The left branch is a leaf whenever the left leaf count is exactly one. The same holds for its right sibling, whose leaf count is found as the difference of that below the internal node and that in the left branch. Whenever a leaf is reached, the next leaf record is read from the separate preorder list and the traversed leaf count is incremented. When pruning a branch, the count of traversed leaves is similarly incremented by the leaf count within that branch. If the left child branch represents an internal node, this will be the *next* internal node by dint of preordering. Similarly, if the right branch represents an internal node, this will be a number of internal nodes further on equal to the leaf count in the left branch. The left and right branches are therefore immediately located, whether external leaves or internal nodes.

4.6. The Automatic Generation of Efficient Bounding Volume Hierarchies

A CSG object defines its own canonical bound hierarchy. In general however there is no immediately obvious choice of hierarchy to build from the bounds of scene objects or a

polyhedron's facets. Many different binary hierarchies may be built from the same leaves, incurring differing computational cost when traversed during ray tracing. An exhaustive search for the most efficient hierarchy is infeasible due to the large number of choices for any non-trivial leaf count, and is impossible anyway without any criterion for assessing efficiency.

4.6.1. Previous Attempts at Automatic Generation

To date the issue has often been avoided altogether. Hierarchies built by scene modellers may yield some increase in the efficiency of synthesis. These tend to be sub-optimal however, being built specifically to simplify modelling rather than image synthesis. Better hierarchies which seem intuitively efficient have been employed.

Various median-cut construction algorithms have been described for binary trees, with clipping planes recursively partitioning a group of leaf bounds [Kay,Kajiya;1986]. However, such constructions tend simply to balance leaf count between branches rather than weighting by expected cost. Consider a scene containing an isolated big object close to the viewer and several little objects grouped distant from the viewer. A binary branching by median object cut at the root would yield one large bound containing the big object and half the little ones, and a smaller bound containing the remaining little objects. The former bound would be struck by most primary view rays and so provide limited computational savings. The latter would often be missed but this query would reject only half the little objects. A better root branching scheme would be to partition the scene with the big object isolated in its own branch and all the little objects grouped in the other. The former branch's bound would still be struck by many rays, but less than before being smaller. The latter branch's bound would usually still be missed, allowing all the little objects to be rejected with a single early query. Clearly, an optimal or quasi-optimal hierarchy should weight for the low expected query cost even after bounding all the little objects together. A construction algorithm has been published which attempts to allow for these weightings with an efficiency metric [Goldsmith,Salmon;1987]. This does not exploit clipping plane inheritance, but generates trees of variable branching ratio. This complicates the derivation

of an efficiency metric which is taken as a loosely justified heuristic. A more rigorously derived metric measuring the expected cost of ray tracing a given hierarchy is desirable to identify the optimal or a quasi-optimal choice.

4.7. The Optimality Condition and the Huffman Tree Parallel

4.7.1. An Efficiency Metric on Bounding Volume Hierarchies

The traversal of a hierarchy incurs costs in bound queries. These costs may be measured in units of six clipping plane intersections. A unit cost is incurred by the initial query of the root bound. If missed, there are no more costs within the hierarchy. Otherwise, an optimal hierarchy will incur *minimal* expected further cost. When the traversal of a hierarchy reaches a given node, the bound of that node is known to be struck as is that of the node's parent, the parent's parent and so on up to the root. As ancestor bounds in object and polygon hierarchies are strict supersets, the first statement is the strongest and implies the others. Both branch bounds are queried. In general this incurs a unit cost of six clipping plane intersections when exploiting clipping plane inheritance. No further traversal cost is incurred from a branch whose bound is missed or is found to be a leaf. Strictly speaking, the traversal also prunes any branch whose bound is struck but only beyond the significant path length interval. However, this case is difficult to allow for in a static efficiency model since the significant interval changes dynamically. Otherwise, the traversal descends down the branch. This generates a recurrence relation for the expected cost over the entire hierarchy. The relation may be simplified to an expression which is proportional to the sum of ray intersection probabilities over the internal node's bounds. [Fig 4.7.1a]. Such probabilities are difficult to predict exactly. For a given ray source and uniform direction distribution the probability of a given bound being struck by a ray will be proportional to the solid angle subtended at the source. This is in turn approximately proportional to the bound's surface area for a distant ray source. Surface area therefore provides a heuristic measure of the probability of an arbitrary ray striking a given bound [Goldsmith,Salmon;1987]. The probability sum to be minimised by an optimal hierarchy

Fig 4.7.1a: An Efficiency Metric on Binary Bounding Volume Hierarchies

The Recurrence Relation for an Efficiency Metric

The expected cost incurred at any node reached during the traversal of a bound hierarchy for a ray of arbitrary source and direction is

$$\text{Cost}(N) = \begin{cases} 0 & \text{For a leaf} \\ 1 + \sum_{B \text{ branch } N} P(B \mid N)\text{Cost}(B) & \text{For an internal node} \end{cases}$$

where

Cost(N) = Expected cost incurred at node N in units of six clipping plane intersections
P(N) = Probability of an arbitrary ray striking the bound of node N

and 'B branch N' denotes a branch of the node. Since any branch's bound is a subset of the node's

$$P(B \mid N) = \frac{P(B \cap N)}{P(N)} = \frac{P(B)}{P(N)}$$

The Expansion of the Recurrence Relation

Given a binary hierarchy, this recurrence for expected traversal cost at any node expands to

$$\text{Cost}(N) = \frac{1}{P(N)} \times \sum_{I \text{ internal } N} P(I)$$

where 'I internal N' denotes an internal node in N's subtree. This may be shown by induction on the hierarchy structure.

Induction Base: A single node tree

$$\text{Cost}(N) = 0 = \frac{1}{P(N)} \times \sum_{\text{empty sum}}$$

Induction Step: A multi-node tree

Assume the hypothesis holds for all trees with fewer leaves than below this node.

$$\text{Cost}(N) = 1 + \sum_{B \text{ branch } N} P(B \mid N) \times \text{Cost}(B).$$

$$= 1 + \sum_{B \text{ branch } N} \left[P(B \mid N) \frac{1}{P(B)} \sum_{I \text{ internal } B} P(I) \right]$$

$$= \frac{P(N)}{P(N)} + \sum_{B \text{ branch } N} \left[\frac{1}{P(N)} \sum_{I \text{ internal } B} P(I) \right]$$

$$= \frac{1}{P(N)} \times \sum_{I \text{ internal } N} P(I)$$

An appeal to induction shows the hypothesis to hold for any tree.

Applying this to the root, an optimal hierarchy will therefore minimise $\sum_{I \text{ internal Tree}} P(I)$

Fig 4.7.1a

may therefore be taken to be approximately proportional to the summed surface area over the internal nodes' bounds. The calculation of a box bound's surface area requires only two multiplications and additions, barring an insignificant constant factor of two [Goldsmith,Salmon;1987]. The terms *larger* and *smaller* are applied interchangeably hereafter to a hierarchy node and the surface area estimate for the probability of an arbitrary ray striking its bound.

4.7.2. The Huffman Tree Parallel

An optimal or quasi-optimal bound hierarchy should minimise bound query count, and so behave like a *Huffman encoding tree* for data compression [Huffman;1952].

The leaves of a Huffman data compression tree hold the distinct data from a given set to be compressed. These are encoded as symbol strings of variable length describing the path from the root to the leaf position. Each encryption symbol defines the direction taken at the next internal node fork. These strings are decoded by filtering down from the root as dictated by each symbol until a leaf is reached. The datum encoded there is output, and the traversal returns to the root with *no need* for an encryption termination symbol. Huffman has described an elegant construction for such a tree which minimises average string length per encoded datum. An optimal bound hierarchy should similarly minimise the average bound query count per ray.

Admittedly, the two hierarchies are not completely analogous. Exactly one path is followed at each internal node of a Huffman tree, as the paths partition the encoded data set. In general a subset of the paths is followed from an internal node in a bound hierarchy since none, one or many branches may remain significant. Four cases of recursive traversal may occur for a binary bound tree compared to two in a Huffman compression tree. Some parallel is evident none the less, and a little analysis on the efficiency metric for a Huffman tree yields the same optimality condition as for a bound hierarchy [Fig 4.7.2a].

Fig 4.7.2a: An Efficiency Metric on Huffman Data Compression Trees

The Recurrence Relation for an Efficiency Metric

The derivation of an efficiency metric on a Huffman tree is analogous to that given for a bound hierarchy [Fig 4.7.1a]. A leaf node has no branches, and hence requires no encryption routing symbol. An internal node incurs a single symbol cost, and the probability of an arbitrary leaf datum being held in a given branch is known beforehand. The average number of encryption symbols needed to differentiate the strings below a given node is therefore

$$\text{Cost}(N) = \begin{cases} 0 & \text{For a leaf} \\ 1 + \sum_{B \text{ branch } N} P(B \mid N)\text{Cost}(B) & \text{For an internal node} \end{cases}$$

where

Cost(N) = Average number of symbols needed to differentiate strings below node N
P(N) = Probability of an arbitrary string being below node N

and 'B branch N' denotes a branch of the node. Notice that since the strings in any branch are a subset of those in the node's entire subtree

$$P(B \mid N) = \frac{P(B \cap N)}{P(N)} = \frac{P(B)}{P(N)}$$

The Expansion of the Recurrence Relation

This recurrence relation is isomorphic to that for an efficiency metric on a bounding volume hierarchy, and the same conditional probability rule holds. An identical inductive argument therefore applies, yielding the expansion

$$\text{Cost}(N) = \frac{1}{P(N)} \times \sum_{I \text{ internal } N} P(I)$$

Applying this to the root, an optimal data compression hierarchy will therefore minimise

$$\sum_{I \text{ internal Tree}} P(I)$$

4.8. The Implementation of a Generalised Huffman Construction

Huffman's construction of an optimal data compression tree maintains a pool of tree nodes. Each node represents a branch of encoded strings and is assigned the probability of an arbitrary string from the encoded data set being amongst these. The pool is initialised with one node per leaf datum. For a binary tree the two nodes merging to the least probable composite are located and removed from the pool. They are combined to a single branch which is marked with the associated probability and returned to the pool. Population partitioning ensures that any composite's probability is the sum taken over the branches:

$$P(N) = \sum_{B \text{ branch } N} P(B)$$

The nodes to merge will always be the two least probable. The process is iterated until only one node remains. This is the root of the hierarchy thus built.

The algorithm may be generalised to provide a construction of quasi-optimal bounding volume hierarchies for solving the scene model in ray tracing. Complete optimality is no longer guaranteed, essentially since only a weaker probability inheritance holds. The probability of a node's bound being struck by an arbitrary ray is only known to be no less than that for any branch:

$$P(N) \geq \max_{B \text{ branch } N} P(B)$$

Under these conditions, the early stages of Huffman's iterative construction of the least probable internal node can rob later internal nodes of the chance to yield a hierarchy of lower average query cost. However, there are generally many worse choices of nodes to merge at any stage of construction. Whilst trees so built by this generalised Huffman construction may be sub-optimal they are rarely catastrophically so.

This thesis derives a novel generalisation of Huffman's construction for bounding volume hierarchies.

4.8.1. Data Structures of Huffman's Generalised Construction

Huffman's construction is traditionally implemented by linking the active pool nodes in a list of ascending probability order. When the strong probability inheritance holds, the

composite node of minimal probability is simply the combination of the nodes at the list head. Unfortunately this does not carry over to bound hierarchies where only the weaker inheritance holds. Two small bounds of low probability may yet yield a large composite of high probability if spatially distant. Though the inheritance condition is weaker, it may still be exploited to reduce tree construction costs.

4.8.2. Initialisation of Huffman's Generalised Construction

A pool of representative nodes is initialised, in which each holds the following records:

- The bound - a set of clipping plane extremes and the surface area estimate of the probability of a strike by an arbitrary ray;
- The list link - a link to the next largest node;
- The *minimal merge partner* - the index from all larger nodes to that yielding the smallest composite and this surface area;
- The leaf count - the number of leaves held below the represented node;
- The index of the represented node - in the leaf list when the leaf count is unitary, or the bound hierarchy otherwise.

The pool is initialised with the given bounds and linked in ascending order. The nodes may be conveniently ordered through memory with a library sorting routine and then directly linked by ascending address. The initialisation of each node's minimal merge partner requires a judicious implementation. An exhaustive search would require the consideration of all larger nodes, and hence a total cost dominated by the square of the node count. A better approach is to consider each larger node in ascending order as a candidate for the minimal merge partner. The order is followed through the list links. A record is maintained of the larger node yielding the smallest composite to date, and this surface area. Should the current candidate node ever prove larger than this, then by the weaker probability inheritance condition this candidate *cannot* be the minimal merge partner. Moreover, the same holds for all subsequent candidates by transitivity since the pool is linked in ascending order, and so the task is complete. The only extra requirement of this over an

exhaustive search is that candidates are considered in a given order. Whilst the efficiency of this search will depend on the given bounds, costs will be no greater and generally lower than for an exhaustive search given an appropriately ordered list.

4.8.3. Iteration of Huffman's Generalised Construction

The hierarchy is built bottom-up by repeated formation of the smallest possible composite from all active nodes. This is the composite formed by the node of smallest minimal merge with its partner, and is found in a search starting at the smallest active node. The search proceeds through the ascending linked list maintaining a record of the node with the least minimum merge to date. Should the current node ever be larger than the least minimum merge, then by probability inheritance again this *cannot* yield the overall minimum composite. By transitivity the same holds for all subsequent candidates and so the task is done.

This node is merged with its minimal merge partner to yield the minimum overall composite. The nodes to be merged are removed from the linked list, combined, and the composite is placed in one of the freed slots. The composite must then be inserted into the linked list *preserving* ascending order.

The insertion point may be found without resorting to an exhaustive search. Each composite node formed is always the smallest possible at any stage. The composites are therefore produced in monotone non-decreasing order. Should the insertion point be after a previously formed composite, it *must* therefore be after the *immediately* previous composite. Otherwise, the insertion point will be after a leaf representative, for there is no other type of node. If the pool is initialised to ascending order through memory, the largest of all active leaf nodes smaller than the composite may be found by repeated address bisection. The node after which any composite is to be inserted is then found by comparing the previous composite with the leaf candidate. The former is already known being the last composite formed, and the latter is found with the bisection of logarithmic complexity. The composite is inserted into the linked list after the larger of these.

Before forming the next hierarchy node, the minimal merge partner records must be maintained. A complete repeated search is *not* required for each node. Indeed, it is only necessary for the new composite and any node whose previous minimal merge partner was one of the deleted nodes. Great savings are made here over construction by repeated exhaustive search.

The only status change for nodes not robbed of their previous minimal merge partners by list deletion is the appearance of the new composite. This is a new minimal merge partner candidate. In such circumstances, only a *single* comparison of the previous minimal merge against that with the new composite candidate is required, updating the minimum merge partner record appropriately. By definition minimal merge partners are selected only from larger nodes. The minimal merge partner of any node larger than the new composite will not have been deleted, nor will the new composite be a valid candidate. The minimal merge partner is therefore unchanged in this case. Similarly, the partner of any node larger than both deleted nodes cannot have been deleted. The new composite is however a new valid candidate over all smaller nodes and is taken into account with a single comparison. The same holds for any node smaller than a deleted node whose previous minimal merge partner has not been deleted.

A complete repeated minimal merge partner search is only necessary for a limited number of nodes. These are the new composite and those nodes smaller than a deleted node whose previous minimal merge partners have been deleted. This search is performed as described before [Section 4.8.2].

4.8.4. Producing the Preorder Hierarchy

As each composite pool node is formed, a parallel node is created storing the bound node in the previously described data structure [Section 4.4] together with left and right branch pointers to the merged nodes. When the pool has been destructively reduced to a single node, the parallel hierarchy is traversed in preorder. The bound data are output without the branch pointers. The leaf nodes are permuted to tree-order and output in a second list, allowing any subsequent tree traversal to locate the correct leaves [Section 4.4]. The

whole construction is performed at a preprocessing stage prior to image synthesis.

4.8.5. The Reduced Construction Costs of Super Hierarchies

To reduce construction costs, bound hierarchies may be built as *super hierarchies*. These are constructed from previously built hierarchies rather than objects or polygons. Super hierarchies are held in the same format and order as described above and are therefore queried identically.

The cost of hand drawn animation is traditionally reduced by overlaying images of dynamic foreground objects over a static background scene. The latter need only be drawn once to be used over an entire sequence, reducing total production cost. Relatively large cost may be justified for backgrounds used over many frames since this results in only low unit cost *per frame*. This technique is not directly applicable to computer generated animation of high realism, as simple overlay does not synthesise shadows, reflections or refractions between foreground and background objects.

However, an animated scene may still be partitioned into groups of dynamic foreground and static background objects to reduce total costs. Hierarchies may be built from such groups, which are then themselves combined in a super hierarchy. The static background hierarchy need only be built once, justifying even a high overhead cost for long sequences. Dynamic foreground hierarchies are re-built for each frame, but generally at low cost for low count. The majority of a scene is often static. Whilst a query of such a hierarchy may be less efficient than that generated from all dynamic and static objects, lower total costs may still result from a more efficient construction.

4.9. Drawbacks of Bounding Volume Hierarchies

Though conceptually simple, bounding volume hierarchies are prone to various drawbacks. These arise from both the characteristics of individual bounds and hierarchies thereof.

4.9.1. Drawbacks of Individual Bounding Volumes

The geometry of a bounding volume must be simpler than that of its contents to provide any savings in a ray query. As a result simple convex bounds have been almost universally employed, typically the sphere [Whitted;1980], world axes-aligned box [Goldsmith,Salmon;1987] or extent slab intersection [Kay,Kajiya;1986]. Whilst the latter may be tailored to an arbitrarily close fit of convex contents, any convex bound will necessarily over-approximate non-convex contents [Section 3.5.2]. Bounds of better fit may be envisaged in such cases but tend to be more complex geometries and are therefore of little practical use [Section 3.5.2].

4.9.2. Drawbacks of Hierarchies of Bounding Volumes

A hierarchy of bounds can only decompose a scene to the level of the leaf bounds from which it is built. Such hierarchies therefore tend to stop at an object's *body* rather than continuing down to the more salient *surface* [Section 3.5.2]. The degree of decomposition offered is inflexible. In general, bounding volume hierarchies are fully constructed before image synthesis offering no opportunities for savings from lazy construction [Section 3.5.2].

Optimal hierarchies prove difficult to recognise before image synthesis, let alone construct. Attempts to define an efficiency metric on such hierarchies rely on assumptions and subjective heuristics to a greater [Goldsmith,Salmon;1987] or lesser [Section 4.6] degree. The justification of these is sometimes questionable. For example, a bound's surface area is taken as an estimate for the probability of being struck by an arbitrary ray from the view model. No matter how large however, any bound entirely behind the source of all such rays with respect to direction will always be outside the significant path length interval and struck by none.

Such hierarchies are therefore prone to over-approximate a region's significance to the scene model solution for a given ray. This arises from the over-approximation of both the contents and the probability of being struck by an arbitrary ray.

Efficient hierarchies are difficult to construct. The cost of any construction attempting to recognise dependencies between bounds tends to be polynomial rather than linear in object count. Where possible the described construction of a quasi-optimal hierarchy avoids the exhaustive search incurring such expense, but this remains the general catch-all. The cost of hierarchy construction therefore tends to grow rapidly with object count despite efforts of restraint [Section 4.8].

Bound hierarchies are also irregular in structure and therefore offer no opportunities for navigation with efficient incremental arithmetic.

The decomposition of a scene by a regular partition avoids many of these problems [Section 3.4.3]. The simplest such partition is the regular grid or *spatial enumeration* [Fujimoto et al; 1986] and is addressed in chapter five.

Chapter 5: Grid Partitions

Synopsis:

Chapter five addresses scene decomposition by a grid partition. Both the navigation and generation of the grid partition are considered. An efficient construction of the grid partition is presented which is based on the octtree. This has been fully implemented and used extensively for realistic image synthesis.

5.1. The Simplification of the Scene Model with Grid Partitions

A scene grid partition or 'spatial enumeration' is a 3D generalisation of a 2D raster screen [Fujimoto et al;1986: Marsh;1987]. Whilst such a pixelated screen may display a globally complicated image containing high detail, possibly from video capture, it is locally simple with each pixel set to a single colour. Similarly, a 3D grid partition of a rectangular box containing many global objects yields local *voxel* regions of greatly reduced significant object count [Section 3.4.3]. An object is significant to the scene model's solution within a voxel when a section of its surface passes through that voxel. Such an object is said to be heterogeneous with respect to the voxel.

The scene model is locally simplified by the voxelation of a containing box [Fig 5.1a]. This simplification may be exploited to reduce solution cost greatly. The box is first queried for ray intersection. If missed, none of the scene is significant and the model is solved trivially. Otherwise, the ray entrance point is transformed from the world to a local coordinate system in which each voxel is a unit cube. The voxel containing this point is identified as the first navigated by the ray. Subsequent voxels are found by navigating the ray through the grid in path length order [Fig 5.1b]. The heterogeneous objects in each voxel are considered for the scene model's solution. The navigation terminates when no voxels remain or earlier if a voxel is reached beyond the significant path length interval. Any object heterogeneous to the first voxel encountered beyond the significant interval has either been considered already or is insignificant to the scene model solution. Moreover, since the ray is navigated through voxels in path length order, any subsequent voxels are beyond the significant interval. By transitivity, their associated objects may also be ignored.

The grid partition is therefore exploited to trade costs of object query for ray navigation between voxels.

5.1.1. Avoiding Repeated Object Query

Each object occupies its own leaf bound in a bounding volume hierarchy [Section 4]. In a grid partition however, one object may be heterogeneous over several voxels. The

Fig 5.1a: Non-Empty Voxels in Cut-Aways of Grid Partitions for Two Scenes

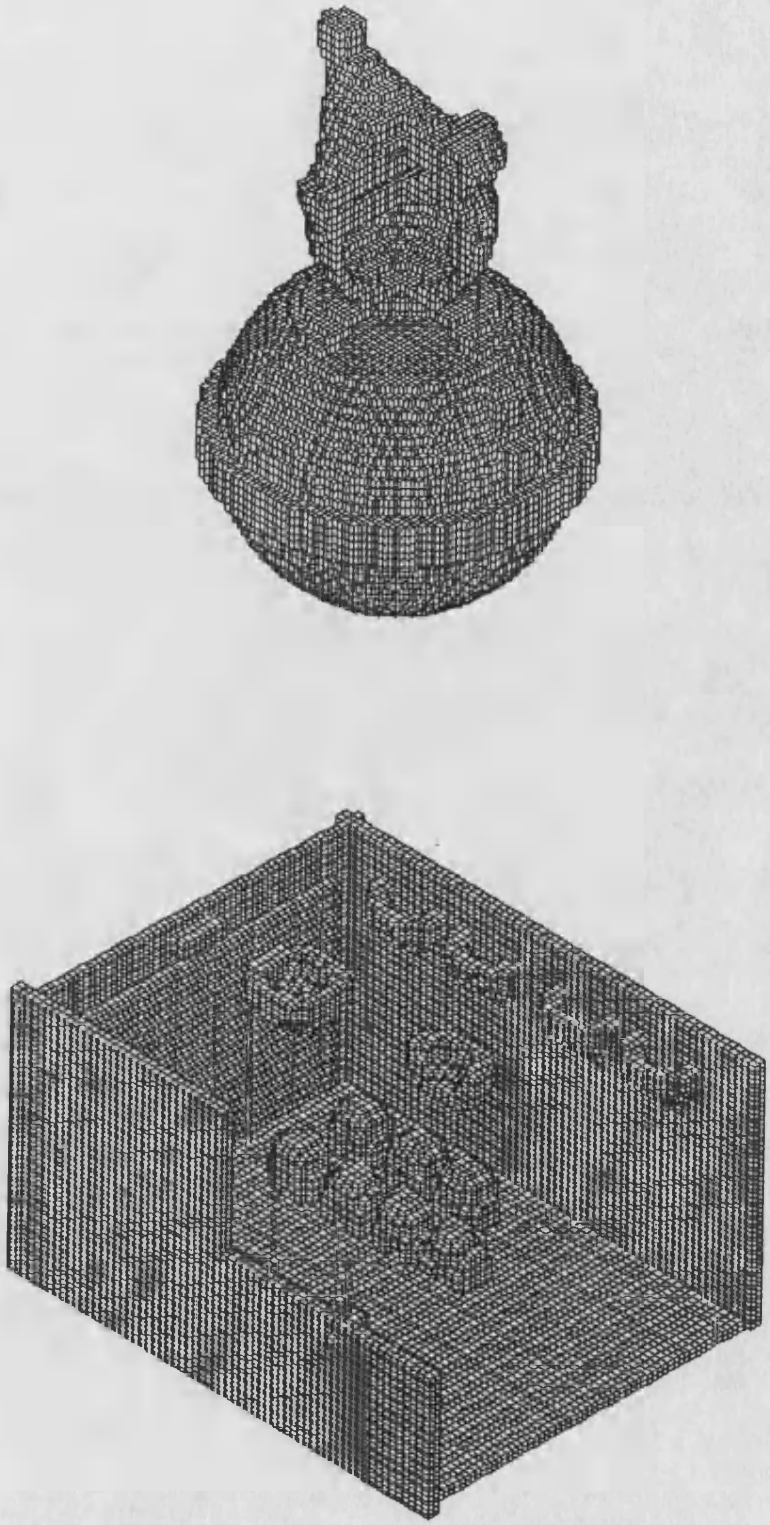
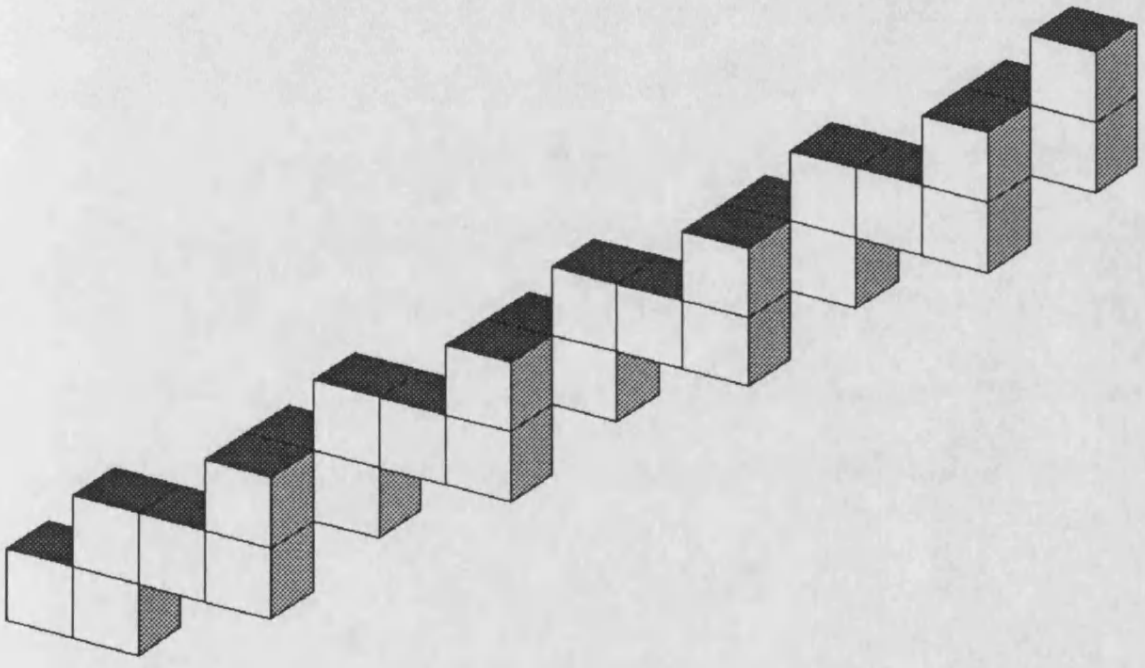


Fig 5.1a

Fig 5.1b: Ray Navigation through the Voxel Cells of a Grid Partition



surfaces of extended cuboids modelling the walls, floor and ceiling of an indoor scene may pass through many voxels. However, each object should only be queried once during the scene model solution. Repeated querying of an object over voxels is clearly inefficient, as the same result is obtained each time. Worse still, repeated querying by illumination rays can produce errors in the shadow cast by a transparent object. In reality such an object has a single chance to cast a shadow. Allowing for the object several times would wrongly increase its shadowing by the corresponding factor. Repeated object query must be avoided.

This is achieved by assigning each object a record of the previous querying ray, in much the same manner as each light source is assigned a record of the previous shadowing object [Section 3.2.7]. The record does not reference a ray by source and direction but rather by an integer count. A total count of all rays traced to date is maintained. This is incremented each time the scene model is solved for another view or illumination ray. Before querying an object in a voxel's heterogeneous list, this record is checked against the current ray count. If equal, the object has already been considered for this ray and is not queried again. Otherwise the object must be queried, after which this record is updated to the current ray count so avoiding any subsequent repetition of the query.

5.2. Previous Grid Partition Navigation Algorithms

The grid partition trades scene model solution costs in object query for ray navigation. This immediately begs the question of how to navigate a ray between voxels in increasing path length order. The navigation should be as efficient as possible to maximise any benefit.

Voxels could be grouped together and navigated as a bounding volume hierarchy [Section 4]. However the grid partition already has a regular structure which may be exploited in a more efficient navigation. Line generators for a 2D pixelated raster screen based on the DDA or *Differential Digital Analyser* may be generalised for ray navigation through a 3D voxel grid [Fujimoto et al;1986: Marsh;1987].

5.2.1. The Differential Digital Analyser

The path between successive pixels or voxels visited by a 2D line or 3D ray respectively is assumed to be navigated through a shared face. This involves no loss of generality, since diagonal steps may be filled in with appropriate intermediary regions sharing a face with both the pre and post step regions.

In the 2D case the next pixel navigated by a line of arbitrary direction will be one of the current pixel's four shared-face neighbours. The direction of any line is constant and falls into a characteristic quadrant. This is easily found and reduces the number of candidate pixels to just two. The 2D DDA identifies the axis spanned at the greatest rate as the *driving* axis [Fujimoto et al;1986]. The other is called the *passive* axis. Lines are generated in pixel width steps along the driving axis. During any step along the driving axis at most one step can occur along the passive axis. Taking the former to go along and the latter to go up, the navigation must then choose between two step cases. These are *along* & *up, along*. The 2D DDA maintains a variable whose sign decides between these. This is known as a decision variable or control term. It is maintained across steps by a recurrence relation involving only the addition of predetermined increments.

5.2.2. Bresenham's Enhancement

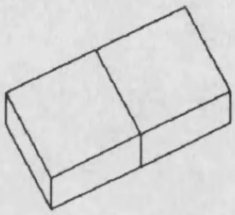
Whilst these increments are constant, their calculation involves division. This inconvenience generally necessitates floating point arithmetic to maintain accuracy even when lines are originally specified over the integers. Moreover, in some cases DDA's involve a degenerate division by zero which has to be handled as an exception. Bresenham's algorithm [Bresenham;1965] multiplies out all such division from a DDA whilst leaving the decision variable's sign unchanged. This yields an efficient line generator for raster screens maintained by additive integer arithmetic with no exception handling. The DDA for 2D line generation through a pixelated screen and Bresenham's enhancement may be generalised for 3D ray navigation through the grid partition of a voxelated box.

5.2.3. The Generalisation of the DDA for 3D Ray Navigation

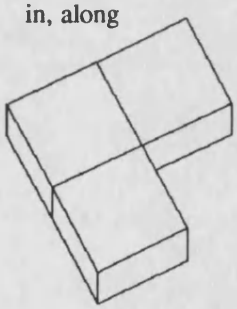
Generalisations of the basic 2D DDA to 3D have been developed by other researchers [Fujimoto et al;1986: Marsh;1987]. The next voxel navigated by a ray of arbitrary direction will be one of the current voxel's six shared-face neighbours. Ray direction is constant and falls into a characteristic octant. This is easily found and reduces the number of significant voxels to three. ARTS, the Accelerated Ray Tracing System, navigates rays with two simultaneous 2D DDA's and so maintains two decision variables [Fujimoto et al;1986]. The axis spanned at the greatest rate is identified as the common driving axis. The other two are passive. Rays are navigated in voxel width steps along the driving axis. During each such step at most one step can occur along each passive axis. Taking the driving axis to go along and the passive axes to go up and in, the navigation must decide between five step cases. These are *along* ; *in, along* ; *up, along* ; *in, up, along* & *up, in, along* [Fig 5.2.3a]. Each decision variable is set to the difference of two distances measured along the driving axis. This is the *passive* distance to the next intersection with a partition plane normal to the associated passive axis minus the *driving* distance to the next partition plane normal to the driving axis. Since rays are navigated in voxel width steps along the driving axis only the passive distance changes. A non-negative decision variable indicates that the ray should *stay* in the same passive partition over the driving axis step. The passive distance changes by a voxel width decrement. The decision variable is maintained by the subtraction of this width. Otherwise, a negative decision variable indicates that the ray should *step* across the passive partition during the driving axis step. The next plane normal to the passive axis is then one further on. The passive distance changes by an increment of the reciprocal ray slope along this axis minus the voxel width. The decision variable is maintained by the addition of this amount. The sign of each variable therefore indicates whether the ray should stay or step along the associated passive axis. The difference of the two decision variables yields the difference of the two passive distances. When steps occur along both passive axes the sign of this difference indicates their order.

Fig 5.2.3a: The Five Cases of a Driving Axis Step

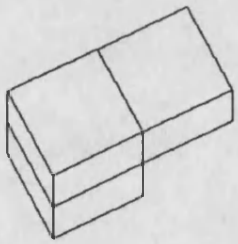
along



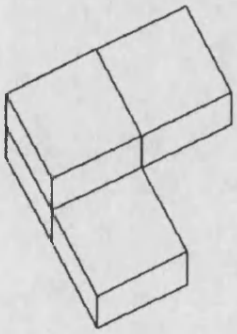
in, along



up, along



in, up, along



up, in, along

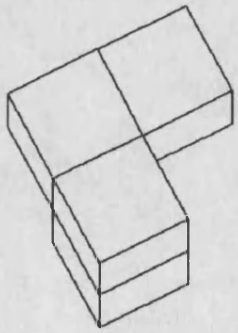


Fig 5.2.3a

This navigation is prone to the division problem of the basic DDA. A degenerate division by zero can arise in the calculation of the reciprocal ray slope along a passive axis, which requires exception handling. This reciprocal is the width of the decision variable's range for the associated passive axis. Whilst each decision variable is bounded within this finite range for any ray of non-zero slope, this range increases without bound for rays of slope approaching zero. The decision variables are therefore *not uniformly* bounded for all rays. The maintenance of decision variables in ARTS navigation was apparently optimised with integer arithmetic. The quantisation method was not published. However, both decision variables would have had to undergo the same quantisation to remain in the same units. This is necessary to ensure that the sign of their difference still indicates the order of steps over both passive axes. The quantisation was probably ray dependent, just fitting the larger decision variable range into a signed integer format. This would complicate quantisation. Moreover the quantisation's resolution would vary between rays, becoming poor for rays with a passive slope approaching zero. The navigation of such a ray would appear to be prone to errors in the exact location of steps along the passive axes. An algorithm has been proposed to allow fixed point arithmetic by splitting decision variables into *integer* and *fractional* parts, but is somewhat convoluted and still requires exception handling [Marsh;1987].

5.3. The Generalisation of Bresenham's Enhancement for 3D Ray Navigation

A generalisation of Bresenham's DDA enhancement for 3D ray navigation would avoid any degenerate division by zero. The decision variables of such a navigation would be uniformly bounded and so have numerically stable quantisation. This thesis derives such a generalisation.

As in ARTS a decision variable is maintained for each passive axis, navigating a *step* or *stay* along that axis over successive voxel width steps along the driving axis. However, these two decision variables are no longer measured in the same units after all division is multiplied out. The order of steps occurring along both passive axes can no longer be deduced from the sign of their difference. An extra decision variable is introduced to

decide this order. This is only examined when steps occur along both passive axes, but is maintained across any driving axis step for future reference. The navigation therefore maintains three decision variables in a *decision vector* which is actually the vector cross product of the ray direction and current ray point within the voxel. The current ray point is moved forward along the ray path from the source during navigation. The ray direction may be scaled to unit *infinity* norm rather than Euclidean norm to fully exploit the resolution of integer quantisation. A vector's infinity norm is the maximum of its components' moduli. This new version of Bresenham's algorithm inherits all the established advantages [Fig 5.3a].

5.3.1. Checking a Voxel is within the Significant Interval

The navigation should terminate as soon as the significant interval is exceeded. A check for this could be made by quantising the interval's maximum limit along the ray path according to the 3D voxelation. A voxel check would then require a comparison along up to all three axes against the 3D quantised maximum. Since navigation is in steps along the driving axis, a better approach is to project the maximum limit down onto this axis and then quantise. This check requires only a single comparison of the 1D quantised maximum against the current driving axis coordinate.

5.4. The Automatic Generation of a Grid Partition

5.4.1. Specifying the Grid Partition to be Generated

Since the structure of a scene grid partition is predetermined, its automatic generation is somewhat simpler than that of a bounding volume hierarchy. Rather than choosing from an enormous range of decompositions, the automatic generation of a grid partition need only select the voxelation's resolution. This may be predetermined by subjective choice. Alternatively it may be increased until either the average number of heterogeneous objects per voxel cell falls below some limit or a maximum resolution is reached. The former termination criterion prevents further decomposition when the scene model has been made locally simple. The latter prevents runaway.

Fig 5.3a: Ray Navigation of a Grid Partition: Bresenham's Algorithm Generalised to 3D

The Navigation Method

A ray is to be navigated in path length order through the cells of a grid partition, from a known point in a known direction. This is achieved in a sequence of cell-width steps along the axis of greatest change, known as the *driving* axis. The other axes are called the *passive axes*. At most one step can occur over each passive axis during each driving axis step.

Each step is navigated according to the component signs of a decision vector. The navigation is maintained with an efficient recurrence relation using constant increments. It is optimised by quantisation to fixed point arithmetic.

Notation

Consider the octant in which the ray's direction falls. In any cell navigated by the ray, this octant contains the vertex intersection of the ray's exit planes from the cell. This vertex is called the *exit vertex.*

Let the ray have a distance vector $\underline{\delta} = (x, y, z)$ of distances to traverse from its current point to the exit planes of the current cell. All distances are strictly positive. This is simply the current ray point vector taken relative to the cell's exit vertex. Let the ray have direction vector $\underline{\Delta} = (X, Y, Z)$ of relative traversal rates across each dimension. All rates are non-negative and the maximum is strictly positive. This maximum corresponds to the driving axis, and is assumed to be the X axis with no loss of generality. The Y and Z axes are the passive axes.

Let each cell have width 'W'.

The ray is iteratively navigated through the grid partition according to a decision vector which is the cross product $\underline{\Delta} \times \underline{\delta}$. The direction vector $\underline{\Delta}$ remains constant throughout navigation. The new value of the distance vector $\underline{\delta}$ after any step is denoted by a tilde superscript, $\tilde{\underline{\delta}}$. The decision vector is maintained by an recurrence relation according to an increment vector $\underline{u}$ such that $\underline{\Delta} \times \tilde{\underline{\delta}} = \underline{\Delta} \times \underline{\delta} + \underline{u}$.

Step Navigation

Each step is navigated through one cell width along the driving axis, traversing exactly one X partition plane. The navigation will only traverse the Y or Z partitions during this step if these planes are struck before the X plane. The component signs of $\underline{\Delta}\times\underline{\delta}$ are examined to determine whether the ray *stays* or *steps* across each partition.

Consider deciding whether the current cell's X exit plane is struck after the Y exit plane. Let 'X before or with Y' and 'X after Y' denote the X plane being struck before or simultaneously with the Y plane and after the Y plane respectively. Let $r_y = y - x\frac{Y}{X}$ be the distance left to traverse across the Y dimension to the Y plane at the intersection with the X plane. Clearly, the X plane is struck before or with the Y plane when this distance is non-negative, so that

$$y - x\frac{Y}{X} = r_y \geq 0 \;\leftrightarrow\; \text{X before or with Y}$$

Now X is known to be positive and since scaling by a positive factor has no effect on sign,

$$Xy - xY = Xr_y \geq 0 \;\leftrightarrow\; \text{X before or with Y}$$

Now

$$\underline{\Delta} \times \underline{\delta} = (X, Y, Z) \times (x, y, z) = (Yz - yZ, Zx - zX, Xy - xY)$$

and so

$$\left[\underline{\Delta} \times \underline{\delta}\right]_z = Xy - xY = Xr_y \geq 0 \;\leftrightarrow\; \text{X before or with Y}$$

The same workings hold for any cyclic permutation of dimensions. The sign of each component of $\underline{\Delta}\times\underline{\delta}$ therefore indicates which of the planes in the other two dimensions is struck first. If positive, the plane in the cyclically proceeding dimension is struck first. If negative, the plane in the cyclically preceding dimension is struck first.

Let 'Y step' and 'Z step' denote respective steps over the Y and Z partitions during a driving axis step. Let the absence of any passive step be denoted 'Y stay' and 'Z stay' accordingly.

The order of any passive steps during the driving axis step is determined through a decision tree requiring up to three component sign examinations of the decision vector :

$$\left[\underline{\Delta}\times\underline{\delta}\right]_z\begin{cases}\geq 0 \rightarrow \text{Y stay; } \left[\underline{\Delta}\times\underline{\delta}\right]_y\begin{cases}\leq 0 \rightarrow \text{Z stay: Step order X}\\ > 0 \rightarrow \text{Z step: Step order Z X}\end{cases}\\ < 0 \rightarrow \text{Y step; } \left[\underline{\Delta}\times\underline{\delta}\right]_y\begin{cases}\leq 0 \rightarrow \text{Z stay: Step order Y X}\\ > 0 \rightarrow \text{Z step; } \left[\underline{\Delta}\times\underline{\delta}\right]_x\begin{cases}<0 \rightarrow \text{Y after Z: Step order Z Y X}\\ \geq 0 \rightarrow \text{Y before Z: Step order Y Z X}\end{cases}\end{cases}\end{cases}$$

The cells navigated during the step are simply those on the other side of the successively traversed planes. Each is located by an associated constant increment of the current index to the cell array. This location is under the caveat of the traversed plane being internal to the partitioned scene box. Otherwise, the navigation of the partition is complete and terminates.

Navigation Maintenance

After a driving axis step there is a new distance vector $\tilde{\underline{\delta}}$ corresponding to the updated ray point in the next driving axis cell. There is therefore a new decision vector

$$\underline{\Delta}\times\tilde{\underline{\delta}} = (Y\tilde{z}-Z\tilde{y},\ Z\tilde{x}-X\tilde{z},\ X\tilde{y}-Y\tilde{x})$$

This is found incrementally from the previous value with an appropriate update vector $\underline{u}$.

Since each step traverses a cell width along the driving axis,

$$\tilde{x} = x \rightarrow \begin{cases} Y\tilde{x} = Yx \\ Z\tilde{x} = Zx \end{cases}$$

Moreover,

$$\tilde{y} = \left\{\begin{array}{l}\text{if Y stay: } y-W\dfrac{Y}{X}\\[2ex] \text{if Y step: } y-W\dfrac{Y}{X}+W\end{array}\right\}$$

and so

$$X\tilde{y} = \left\{\begin{array}{l}\text{if Y stay: } Xy-WY\\ \text{if Y step: } Xy-WY+WX\end{array}\right\};\quad Z\tilde{y} = \left\{\begin{array}{l}\text{if Y stay: } Zy-ZW\dfrac{Y}{X}\\[2ex] \text{if Y step: } Zy-ZW\dfrac{Y}{X}+WZ\end{array}\right\}$$

Similarly

$$X\tilde{z} = \left\{\begin{array}{l}\text{if Z stay: } Xz-WZ\\ \text{if Z step: } Xz-WZ+WX\end{array}\right\};\quad Y\tilde{z} = \left\{\begin{array}{l}\text{if Z stay: } Yz-YW\dfrac{Z}{X}\\[2ex] \text{if Z step: } Yz-YW\dfrac{Z}{X}+WY\end{array}\right\}$$

Fig 5.3a

Therefore

$$\underline{\Delta}\times\tilde{\underline{\delta}} = \begin{cases} \text{Y stay, Z stay: (Yz–Zy, Zx–Xz+ZW, Xy–Yx–YW)} \\ \text{Y step, Z stay: (Yz–Zy–ZW, Zx–Xz+ZW, Xy–Yx–YW+XW)} \\ \text{Y stay, Z step: (Yz–Zy+YW, Zx–Xz+ZW–XW, Xy–Yx–YW)} \\ \text{Y step, Z step: (Yz–Zy+YW–ZW, Zx–Xz+ZW–XW, Xy–Yx–YW+XW)} \end{cases}$$

Conveniently, the X divided terms in the first component cancel to avoid undesirable division.

Therefore

$$\underline{\Delta}\times\tilde{\underline{\delta}} = \underline{\Delta}\times\underline{\delta} + \underline{u}$$

where

$$\underline{u} = \begin{cases} \text{Y stay, Z stay: (0, ZW, –YW)} \\ \text{Y step, Z stay: (–ZW, ZW, XW–YW)} \\ \text{Y stay, Z step: (YW, ZW–XW, –YW)} \\ \text{Y step, Z step: (YW–ZW, ZW–XW, XW–YW)} \end{cases}$$

This update vector is constant in all cases, and need only be calculated once before navigation. The appropriate update vector is applied after the traversal of the navigation's decision tree.

Fixed Point Quantisation to Optimise Navigation

All steps are navigated according to the decision vector $\underline{\Delta}\times\underline{\delta}$ and the update vector $\underline{u}$. The navigation may proceed in fixed point arithmetic if the components of both vectors fit in the appropriate format. Consider a 32-bit signed format as an example, storing integers in the interval $[-2^{31},2^{31})$. All vectors will fit into this format provided their infinity norms are below a critical bound, 2^{31} in this example. A vector's infinity norm is the maximum of its components' moduli, denoted $\|_\infty$. Since $\underline{\delta}$ lies within a cube voxel of width W, then $|\underline{\delta}|_\infty \le W$. Moreover, $|\underline{\Delta}\times\underline{\delta}|_\infty \le |\underline{\Delta}|_\infty\times|\underline{\delta}|_\infty \le |\underline{\Delta}|_\infty W$ and $|\underline{u}|_\infty \le |\underline{\Delta}|_\infty W$. All components will therefore fit as required provided $|\underline{\Delta}|_\infty W < 2^{31}$.

The available resolution is to be split between the ray direction and voxel width. There is no obvious reason for apportioning more resolution to one more than the other, so fourteen bits are assigned to each. Then $|\underline{\Delta}|_\infty, W < 2^{15} \rightarrow |\underline{\Delta}|_\infty W < 2^{31}$. This quantisation is easy when

representing real numbers in a mantissa/exponent floating point format such as the IEEE standard. The direction vector is normalised to unit infinity norm and cell size to unit width by appropriate scalings. The fourteen most significant bits are extracted from each relevant variable's mantissa with efficient bit shifts and masks. Their exponents are of no consequence. This quantisation allows navigation to proceed through the voxelation under only additive fixed point arithmetic. It is incorporated into floating point calculations querying the scene box for ray intersection at the start of the scene model's solution.

Fig 5.3a

5.4.2. Assigning a Heterogeneous Object List to a Voxel

Once the decomposition's resolution is determined, each voxel cell in the grid must be assigned its heterogeneous object list. There are two obvious ways of considering the scene to achieve this. On one hand, the scene may be considered object by object. A given object is then added to the heterogeneous list of every voxel its surface passes through. On the other hand, the scene may be considered voxel by voxel. A given voxel then has its heterogeneous list augmented with every object whose surface passes through it.

5.4.3. The Object by Object Generation of a Grid Partition

The object by object method may appear superficially attractive [Marsh;1987]. A generalisation of a 2D paint system's region fill [Foley,Van Dam;1984] could perhaps reduce the number of voxels to be considered for a connected object's surface. Instead of filling in colour over a pixel region starting from a given pixel seed, an object surface could be filled over a voxel region from a given voxel seed. A recursive fill of this type would spread in a twenty-six connected manner. However, on closer examination this approach has several drawbacks.

A fill from a single seed point can only cover the connected surface containing that point. Whilst each primitive modelled has a single connected surface, more complex CSG objects may have several separated by boolean union or symmetric difference. Whilst a single seed is easily found for a primitive surface, several seeds are needed for such CSG geometries. The general location of these can be difficult.

Ideally each voxel's heterogeneous list should be stored in contiguous memory for ease of future reference. When working object by object a voxel's total heterogeneous list length is not known before the objects are added to it. This complicates storage allocation.

The lazy construction of grid partitions by dynamic generation during image synthesis may reduce construction times. Only those cells navigated by a ray need actually be assigned their heterogeneous object list. The others are never reached and can be ignored. For

example, voxels containing only surfaces on opaque objects which are back-facing with respect to all rays are never reached. In lazy construction each voxel is only assigned its heterogeneous list when first navigated by a ray. Fewer voxels are assigned their heterogeneous lists than in exhaustive preprocessing. Lazy construction requires the heterogeneous object list to be assigned to a known voxel. This is orthogonal to the object by object approach, which augments a known object to voxels' heterogeneous lists. Lazy construction could be attempted with an exhaustive search around the connected surfaces of each object, but this is clearly counter productive.

5.4.4. The Voxel by Voxel Generation of a Grid Partition

The voxel by voxel method suffers none of these drawbacks. A judicious implementation can avoid both an exhaustive object search for each grid cell and the need to consider every cell. The reader may have remarked that a grid partition is not thought of as a hierarchy during traversal above the trivial single level constituted by a scene voxel parent and the many voxel children. However grid partitions are efficiently generated by *octtree* hierarchies [Section 3.4.3].

An octtree generates a partition recursively by the simultaneous bisection of a voxel in each dimension to spawn eight children. Clearly an object can only be heterogeneous to any child if previously heterogeneous to the parent voxel. Heterogeneous object lists therefore form a chain of strict subsets down successive octtree generations of monotonically non-increasing length. The current voxel's heterogeneous object list is remembered throughout the recursion. Only these objects need be reconsidered for each child, reducing the total number of object considerations during generation.

For any non-trivial resolution many grid cells have empty heterogeneous lists, with every object rather in a homogeneous state. The octtree decomposition simultaneously identifies large groups of such cells when they constitute a current voxel with an empty heterogeneous list. By monotonicity all successive voxel generations down to the leaf cells must also have empty lists. Further recursive decomposition is unnecessary. All descendant cells are simultaneously assigned empty lists. This reduces greatly the number

of voxels considered. The generalisation of the quadtree complexity theory [Samet,Webber;1988: Hunter,Steiglitz;1980] to the octtree shows that this number will be bounded by a function proportional to the sum of the maximum decomposition depth and the number of cells at this depth with non-empty heterogeneous lists.

5.5. Data Structures for the Representation of a Grid Partition

Each voxel cell is stored as a single integer index to a global array which itself consists of integers. The integer indexed by any cell holds the length of that cell's heterogeneous object list. These objects are listed in the adjacent block of this length with further indices to a global object array. The storage required at each cell is then constant even though the heterogeneous list length may vary. Any variation is ironed out by indirection to the global array which may store lists of arbitrary length. For readers familiar with the 'C' programming language, this double indirection may be compared to the 'argv' argument of a 'main' function. The grid partition is stored as a 3D array of these cells. Since each cell occupies uniform storage the array is indexed directly by Cartesian coordinates. The global heterogeneous list is constructed during the octtree decomposition.

A bit may be allocated in each cell integer as a flag to lazy construction [Section 5.4.3]. This is initially unset and becomes set only when first navigated by a ray. Any ray navigating the cell checks the bit before considering the heterogeneous object list. If still unset the list must be dynamically assigned since this has not yet been done. The bit is then set as a flag to any subsequent navigation. The heterogeneous list is then known to be assigned and is considered for this cell.

5.6. Storage Considerations

Many cells may reference identical heterogeneous lists in the global array. All cells with no heterogeneous objects reference an empty list constituting a single integer. This indicates a list length of zero. In general many cells will have no heterogeneous objects. If the grid's resolution is sufficient to make the cells significantly smaller than most objects, all the cells in the voxelation of an isolated object's surface will reference an identical list. This

constitutes an integer indicating a list length of one followed by this object's index in the global object list. Other cells may reference identical lists comprising several objects.

Such heterogeneous list duplications are all made to reference a single entry in the global array rather than separate copies. This keeps down memory requirements. Before storing a cell's heterogeneous list in the global array, a check is made against the previous entries. If a duplication is found, the cell is made to reference this previous entry and the global array is unchanged. Otherwise this distinct list is added to the global array at a position then referenced by the cell. The global array is therefore maintained without list duplications.

5.6.1. Checking for Duplications with the Empty List

The empty heterogeneous list is particularly common, occurring for any cell entirely inside or outside every object. An empty list is allocated at a known position at the start of the global array before generating the grid partition. This entry comprises a single integer indicating a list length of zero. Any cell with an empty list is made to reference this entry. A reference to an empty list may then be recognised without indirection during any subsequent navigation of the grid. The majority of cells navigated by a ray to solve the scene model should be empty, since navigation usually terminates immediately on finding a surface intersection within the significant interval. The list duplication check starts with this special case candidate.

5.6.2. Checking for Duplications with a Previous Non-Empty Entry

Most scene objects are spatially coherent. List duplications are therefore common over neighbouring cells. The construction of an octtree recurses over the siblings spawned from a decomposed voxel in some order. This is commonly the *absolute Morton digit* order [Fig 5.6.2a] [Morton;1966]. This order is spatially coherent in that cells which are close in the recursive 1D order of visitation tend to be spatially close in the 3D scene partition [Fig 5.6.2b]. Any list failing the empty list duplication test is non-empty and by dint of this coherency may well be a duplication of the previously allocated non-empty entry. This

Fig 5.6.2a: The Morton Code System

Consider the eight siblings spawned by the simultaneous bisection of a voxel in each dimension. Each octant child is conveniently indexed by its *absolute Morton digit* from the parent [Morton;1966]. This comprises three bits, one for each dimension conventionally in the order ZYX. The parent is split into eight children with three bisection planes, one in each dimension. The absolute Morton digit of each child is defined bitwise. Each bit is set if the child lies above the corresponding bisection plane and unset otherwise. The result absolute Morton digit is denoted *abs_Morton_digit(child).* A relative Morton digit may also be defined for each child with respect to one of its siblings, denoted *rel_Morton_digit(child,sibling).* Each dimension bit is set if the child lies on the opposite side of the bisection plane to this sibling and unset otherwise. Let XOR denote bitwise exclusive or. Then by these definitions

rel_Morton_digit(child,sibling) = abs_Morton_digit(child) XOR abs_Morton_digit(sibling)

Bitwise exclusive or is commutative, which infers the intuitively obvious symmetric relationship

rel_Morton_digit(child,sibling) = abs_Morton_digit(child) XOR abs_Morton_digit(sibling)

= abs_Morton_digit(sibling) XOR abs_Morton_digit(child) = rel_Morton_digit(sibling,child)

Moreover, bitwise exclusive or with a given value is its own inverse. This provides a means of recovering a child's absolute Morton digit from its relative digit

rel_Morton_digit(child,sibling) = abs_Morton_digit(child) XOR abs_Morton_digit(sibling)

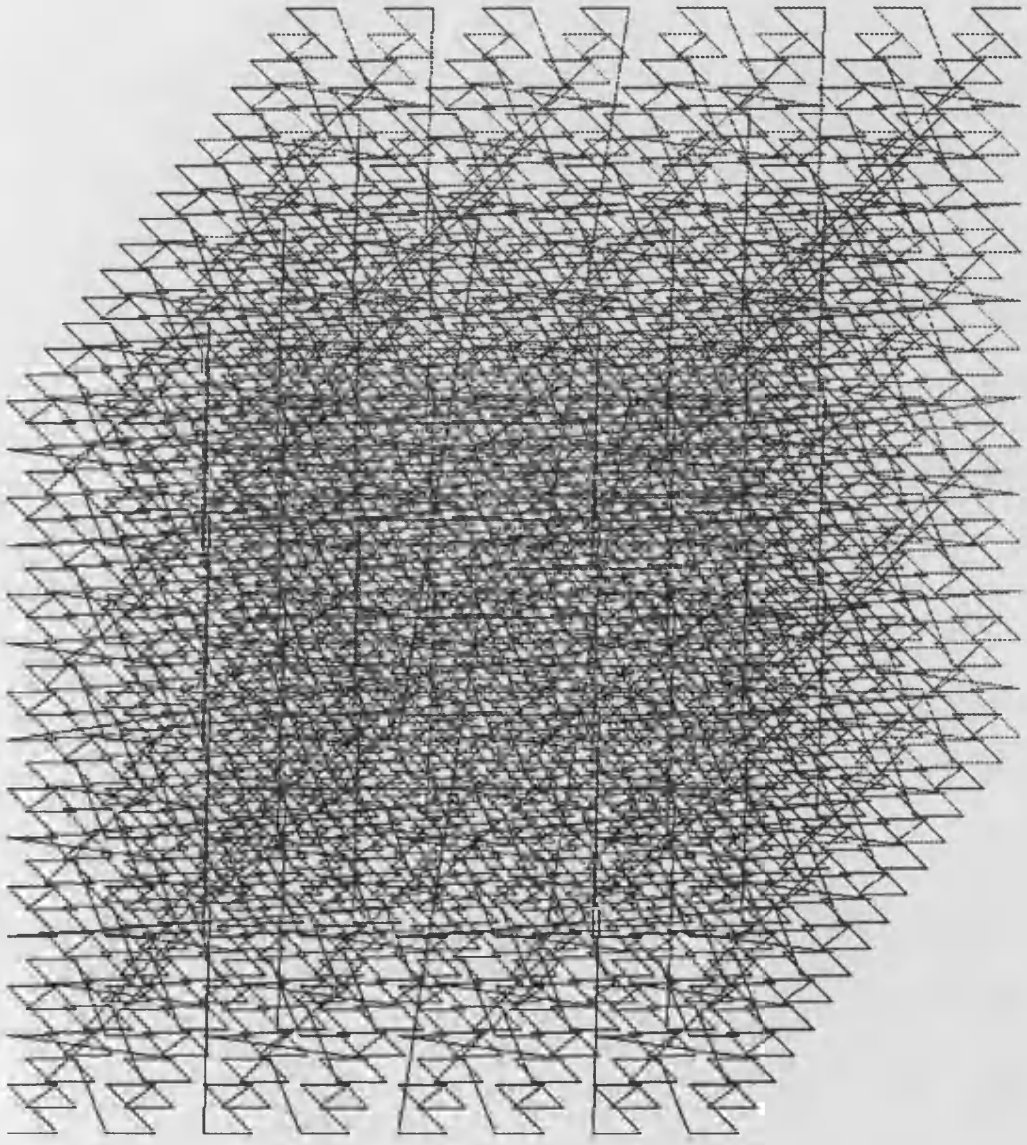
→ rel_Morton_digit(child,sibling) XOR abs_Morton_digit(sibling) = abs_Morton_digit(child)

A Morton digit locates a child voxel down one octtree generation. A similar Morton digit will locate that child's own offspring down another generation. These two digits may be concatenated to locate a grandchild over two generations. Further Morton digits may be concatenated to locate descendants over any number of generations in a string of that number of Morton digits. The digit string identifying any octtree voxel from the root is called that voxel's *Morton code.* Let the vertex in a voxel's octant of absolute Morton digit zero be called the zero vertex. Consider a Cartesian coordinate frame with origin at the root voxel's zero vertex. Each successive digit of a voxel's Morton code may be taken as the absolute Morton digit. The resultant code is called the absolute Morton code. This is easily shown to be the three Cartesian coordinates of the voxel's

zero vertex interleaved together. This interleave is an efficient mapping between 3D Cartesian coordinates and 1D Morton code. Alternatively, each digit may be taken relative to a variable sibling and its bits concatenated in a variable order defined by the previous digit. An appropriate definition will produce a Peano Morton code. This has the characteristic property that any two voxels with subsequent Peano codes share a common face rather than merely tending to be close as in the Morton scheme [Spackman;1987].

Fig 5.6.2a

Fig 5.6.2b: The Morton Order of Visitation for an Octtree Decomposition



observation is exploited by maintaining a record of the previously referenced non-empty entry throughout the decomposition. Any non-empty list is checked for duplication against this record. This provides savings in much the same manner as the record assigned to each light source of the previous opaque object found to cast a full shadow in image synthesis [Section 3.2.7]. The spatial coherency of the octtree's construction may be enhanced by recursing in *Peano digit* order [Fig 5.6.2c]. The Peano ordering is easily followed with bitwise operations such as masks and shifts [Spackman;1987].

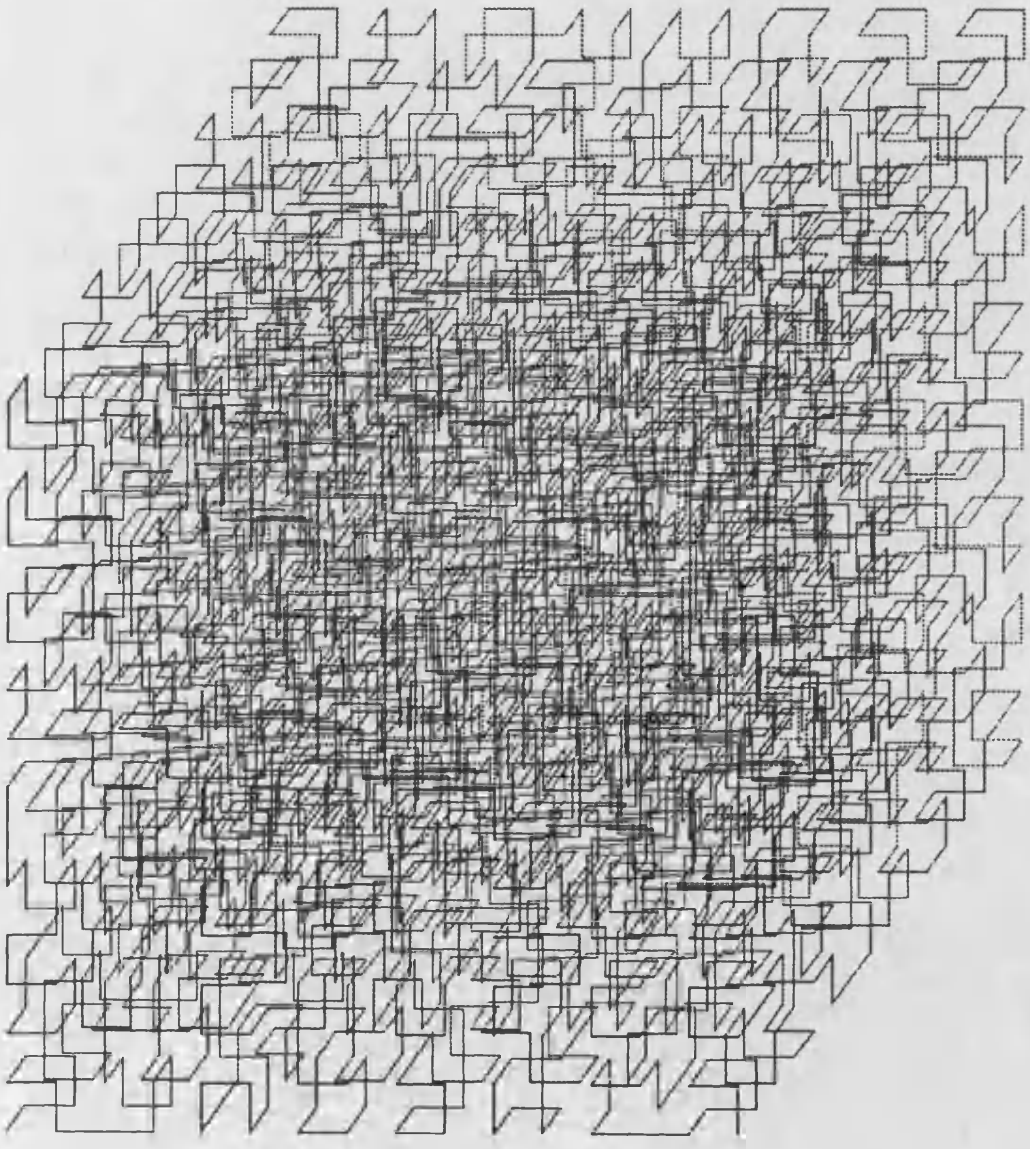
5.6.3. Checking for Duplications in the General Case

If a given list is neither empty nor a duplication of the previously allocated non-empty list, all other distinct entries remain candidates for duplication. A linear search for any such duplication would be inefficient. A binary tree search is preferable.

An ordering is imposed on heterogeneous object lists by defining a concept of difference. If the difference of two lists is negative the first is taken to be the smaller; if positive the second is taken to be the smaller; otherwise, the lists are taken to be equal. List length provides the first ordering key. The difference in two lists' length is easily calculated and is taken as the lists' difference if non-zero. Otherwise, the lists are of equal lengths and the difference in the subsequent pairs of object indices provides the second ordering key. The difference in object indices is calculated pairwise along the lists until either a non-zero value arises or the lists become exhausted. The resulting index difference is taken as the lists' difference, and will be zero only for identical lists.

A binary tree representing all distinct non-empty entries is maintained throughout the construction. A duplication search for a given list is achieved by filtering down the tree according to the ordering. If the difference between the given list and current tree entry is zero, a duplication has been found and the task is complete. Otherwise the search recurses down the left or right branch of the tree depending on whether the difference is negative or positive. The given list can only be distinct from all previous entries if the tree is exhausted without finding a duplication. Such a distinct list is grafted onto the tree after the leaf where the tree became exhausted. The tree is initially empty and is thereby

Fig 5.6.2c: The Peano Order of Visitation for an Octtree Decomposition



guaranteed to be maintained without list duplications.

The efficiency of this search depends on the tree's structure which is determined by the order of the presented data. In the worst case, degenerate presentations such as a monotone ordering result in a linear tree in which one of each pair of sibling branches is empty. Such extreme presentations are rare in practice however, and a random presentation may be expected to produce a reasonably balanced tree. Search times are then logarithmic in distinct list count. Moreover, the earliest distinct lists in the presentation are grafted closest to the tree root. The more frequently occurring lists are therefore most likely to be grafted close to the root, allowing subsequent duplications to be found quickly.

The efficiency of the duplication search could be optimised by dynamic tree balancing to enforce a balanced structure throughout decomposition. Search times would then be logarithmic in distinct list count even for trees built from degenerate presentations, once rectified in this manner. However the tree search generally proves adequate without such balancing, which would of course incur further overhead.

5.7. Testing for Heterogeneity with the Intermediate Value Theorem

The recursive octtree decomposition has been shown to be an attractive means of constructing a grid partition. The construction must allocate a heterogeneous object list to a given voxel. This requires a test for a given object's surface passing through that voxel, which is characterised by the existence of roots to the object's 3D height function within the voxel. A test for the *existence* of roots to an object's *trivariate* height polynomial in 3D space is a somewhat different task to the *location* of roots to the derived *univariate* polynomial along a 1D ray in path length. On one hand, the former task is more difficult due to the extra degrees of freedom in 3D space. On the other, this task is easier due to the weaker requirement of existence rather than location.

Any voxel may be considered as a 3D interval. A 1D interval comprises all points lying between two extremes. These are a lower *infimum* and an upper *supremum* which model the spread of a single variable within these limits. An N-D interval is defined as the

Cartesian product of 'N' such 1D intervals. This models the spread of 'N' *independent* variables. A voxel is a 3D interval modelling the spread of each independent coordinate within a specified 1D interval.

Any object's height function is a continuous map from 3D world space to 1D height. By the intermediate value theorem a sign inversion root of such a function will occur within a 3D voxel interval when the function assumes both a negative and a positive value within that interval. A check for such values is sufficient for a heterogeneity test. In layman's terms a voxel contains part of an object's surface when it contains both points inside and points outside the object. This check need not undertake the more difficult task of actual root location.

5.7.1. Testing with State Codes

An object may be classified as being in one of three possible states with respect to a voxel. If the object's height function is uniformly negative over the voxel, then this voxel is homogeneously inside the object. The object is said to be in HOMO_IN state. Otherwise if the object's height function assumes both negative and positive values over the voxel, then the voxel contains both points inside and outside the object and therefore a section of its surface by the intermediate value theorem. The object is said to be in HETERO state. Otherwise the object's height function is uniformly positive over the voxel which is therefore homogeneously outside the object. The object is said to be in HOMO_OUT state. Any object to be included in a voxel's list is characterised by a HETERO state code, and is tested for heterogeneity via this code.

5.7.2. Attempts at Exact State Code Classification

An object's state may be exactly classified in some special cases. Since any voxel is convex, its intersection with any convex object must also be convex. Suppose a convex object contains all of a voxel's vertices. Then by convexity the object contains all other points within the voxel. Equivalently if no vertex is outside the object then no other voxel points are outside. The convex object is then in HOMO_IN state. Conversely, if some

vertices are outside the object then some voxel points are outside - specifically these vertices. The convex object is then not in HOMO_IN state but rather HETERO or HOMO_OUT state. The existence of voxel points outside a convex object may therefore be deduced from a test restricted to the vertices. There are only finitely many vertices as opposed to the infinite number of voxel points. Each vertex may therefore be tested for being outside the object by evaluating the height function at that point.

The local plane, sphere, cube and cylinder are all convex and the double cone is the union of two convex single cones. Any convex primitive remains convex as a world instance by linear transforms. The intersection of any two convex objects also remains convex. Such convex objects may be exactly classified in the HOMO_IN state.

The local plane also has a convex complement. The existence of voxel points outside this complement and hence inside the plane may be deduced by a similar vertex test. This provides an exact classification of all states for the plane.

However, none of the other primitives has a convex complement. The existence of voxel points inside these primitives cannot be deduced by merely testing each voxel vertex. For example, a cylinder may pass through a voxel yet only intersect the outer faces rather than covering the vertices.

It may be remarked that a voxel contains points inside the world instance of an infinite cylinder or cone when either the object's axis pierces the voxel or a voxel edge is at least partially inside the object. The axis and edges may be considered as rays in 3D world space and the voxel as a cube. The axial ray may be tested for voxel intersection and the edge rays for object intersection as in ray tracing [APPENDIX C].

A voxel has points inside any object when the height function assumes a negative value over the voxel, characterised by a negative minimum. Similarly it has points outside an object when height function assumes a positive value over the voxel, characterised by a positive maximum. The minimum value of an undeformed sphere's height function will occur at the voxel point of minimal displacement in each dimension from the sphere centre. A voxel may be tested for points inside a sphere by testing the height function's sign at this

point.

This piecemeal approach yields exact state classifications in some cases, but is not applicable to a general object. Some primitives are not addressed. For example neither the existence of points inside nor the existence of points outside a torus can be deduced. Some deformations are not addressed for certain primitives. For example the sphere may only be classified in HOMO_OUT state if undeformed. General CSG objects may be only be classified in HOMO_IN state if convex or in HOMO_OUT state if of convex complement. Each test relies on characteristics peculiar to each object and is not readily extensible to other geometries. A more unified framework is desirable within which any object state may be classified.

5.7.3. Approximate State Code Classification

An arbitrary CSG object's state must be classified within a voxel to test for heterogeneity. An exact classification proves difficult when allowing for the many local primitives, deformed world instances and boolean constructions in the CSG scene model. However state codes may be classified by more tractable approximations whose inaccuracies are not catastrophic. These are called *conservative* approximations.

The HETERO state is the most general state, indicating the existence of both voxel points inside and outside an object. Such objects are retained in the voxel's heterogeneous list. The HOMO states are more restrictive, indicating the absence of any voxel points inside or outside the object. Such objects are rejected from the voxel list.

A general state code approximation may be prone to two types of error. A HOMO object may not be recognised as such but rather classified in the more general HETERO state. Whilst this object would be wrongly included in the voxel list, this is a *conservative* or cautious error. It is not catastrophic since the scene model's solution is not affected. Moreover, if the approximation is 'close' in that the object is truly in HETERO state with respect to some neighbouring voxel this need not adversely effect the efficiency of scene model solution. Any ray navigating the voxel may well find that the object has just been

queried or is about to be queried at such a neighbour. No extra cost is incurred in this case since query repetition is avoided [Section 5.1.1]. Alternatively a HETERO object may be wrongly classified as HOMO and rejected from the voxel list. This is a *rash* error which can have a catastrophic effect on image synthesis. Voxel holes may appear in the surface of a HETERO object incorrectly classified as HOMO due to errors in the solution of the scene model.

Provided state codes are approximated conservatively the scene model will be solved correctly. The solution is still efficient if the approximation is close rather than say naively classifying every object in HETERO state whatever.

Conservative approximations can be found whose inaccuracies decrease with voxel size. This generates advantageous feedback into the recursive octtree decomposition. Suppose a conservative error results in a HOMO object being classified as HETERO for a given voxel and being included in its list. The voxel is decomposed into eight children and the object's state is reclassified over each. The approximation is more accurate over this voxel generation since each child is smaller. The object may well be correctly classified in its true HOMO state over each child and rejected from their lists. The decomposition should recognise its earlier mistake and *backtrack* to delete the object from the parent list. This situation is easily recognised by a post recursion check that the object survives in at least one of the children's lists. If not, that object should be deleted from the parent's list. Such a deletion may well render the parent's list empty. The children are then no longer significant and may be deleted to reclaim storage.

5.7.4. Approximation Techniques

A general state code classification may resort to conservative approximation to render the associated mathematics more tractable. For example, each object could be approximated with a pair of geometrically simple inner and outer bounds [Waij,Heckbert,Glassner;1988: Beacon et al;1989]. The existence of voxel points inside or outside an object could then be conservatively inferred from that of points inside the outer bound or outside the inner bound respectively. However, the inaccuracy of this rather naive approximation would be

determined by these bounds rather than voxel size. Ideally any inaccuracy should decrease with voxel size to disappear in the decomposition limit [Section 5.7.3].

An efficient and generally applicable approximation with this property is provided by *interval analysis*. This analysis conservatively estimates the extremes of an object's height function within a voxel.

5.8. Conservative Heterogeneity Tests with Interval Analysis

Interval analysis defines algebra on intervals rather than points [Rall;1981: Moore;1979]. This analysis provides the interval of values assumed by a multivariate continuous function when each argument varies independently within a given interval. The algebra addresses the spread rather than simple point location of variables and was developed to track worst-case rounding errors during digital computation [Moore;1965]. Interval analysis is exact for polynomial functions of independent terms. Such functions can only assume values within the derived interval over their argument intervals, and will assume each value therein. Interval analysis is conservative for functions of dependent terms. Such functions can still only assume values within the derived interval over their argument intervals but may not assume the extreme values.

5.8.1. Interval Analysis of Univariate and Bivariate Arithmetic

Appropriate interval arithmetic is easily derived for the common univariate and bivariate continuous functions [Rall;1981: Fig 5.8.1a]. The interval of an object's height function over a voxel may be evaluated in this arithmetic by recursing over the object's CSG description tree. The height interval determines whether the voxel lies inside, outside or possibly across the surface of the object in the same manner as the function's evaluation in real arithmetic for a scene point. The X, Y & Z coordinates of a point may be substituted into the function which is then evaluated in real arithmetic to a real height. The point is inside, outside or on the surface of the object if this height is negative, positive or zero respectively. The X, Y & Z intervals of a voxel may be formally substituted into this function which is then evaluated in interval arithmetic to a height interval. The voxel is

Fig 5.8.1a: Interval Analysis

Notation

Symbol	Meaning
$\vec{U} = [U_I, U_S]$	1D interval comprising all points between the lower infimum U_I and upper supremum U_S
$0 > \vec{U}$	$\vec{U}$ is uniformly negative : $U_S < 0$
$0 \in \vec{U}$	$\vec{U}$ contains zero : $U_I < 0 < U_S$
$0 < \vec{U}$	$\vec{U}$ is uniformly positive : $0 < U_I$
$\text{centre}(\vec{U})$	Interval centre at $\frac{U_I+U_S}{2}$

Derivation

The algebra derived by interval analysis is illustrated graphically - interval exponentiation by a Cartesian graph, interval product by an enlargement graph, and interval addition and extrema by vector graphs.

Interval exponentiation : $\vec{U}^n$

Visualised as a Cartesian graph

n odd $\rightarrow \vec{U}^n = [U_I,U_S]^n = [U_I^n,U_S^n]$

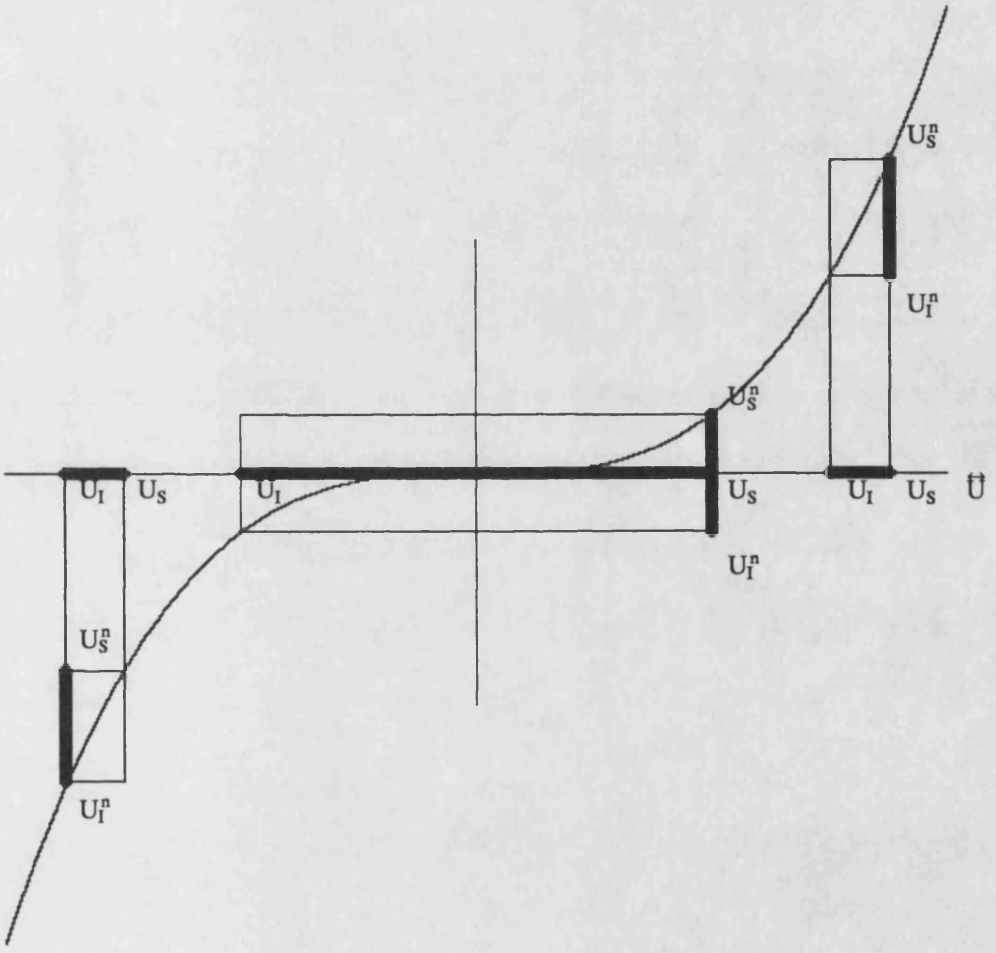


Fig 5.8.1a

Interval exponentiation : $\vec{U}^n$

Visualised as a Cartesian graph

$$\text{n even} \rightarrow \vec{U}^n = \begin{cases} 0 > \vec{U} \rightarrow \vec{U}^n = [U_I,U_S]^n = [U_S^n,U_I^n] \\ 0 \in U \rightarrow \begin{cases} 0 > \text{centre}(\vec{U}) \rightarrow \vec{U}^n = [U_I,U_S]^n = [0,U_I^n] \\ 0 \leq \text{centre}(\vec{U}) \rightarrow \vec{U}^n = [U_I,U_S]^n = [0,U_S^n] \end{cases} \\ 0 < \vec{U} \rightarrow \vec{U}^n = [U_I,U_S]^n = [U_I^n,U_S^n] \end{cases}$$

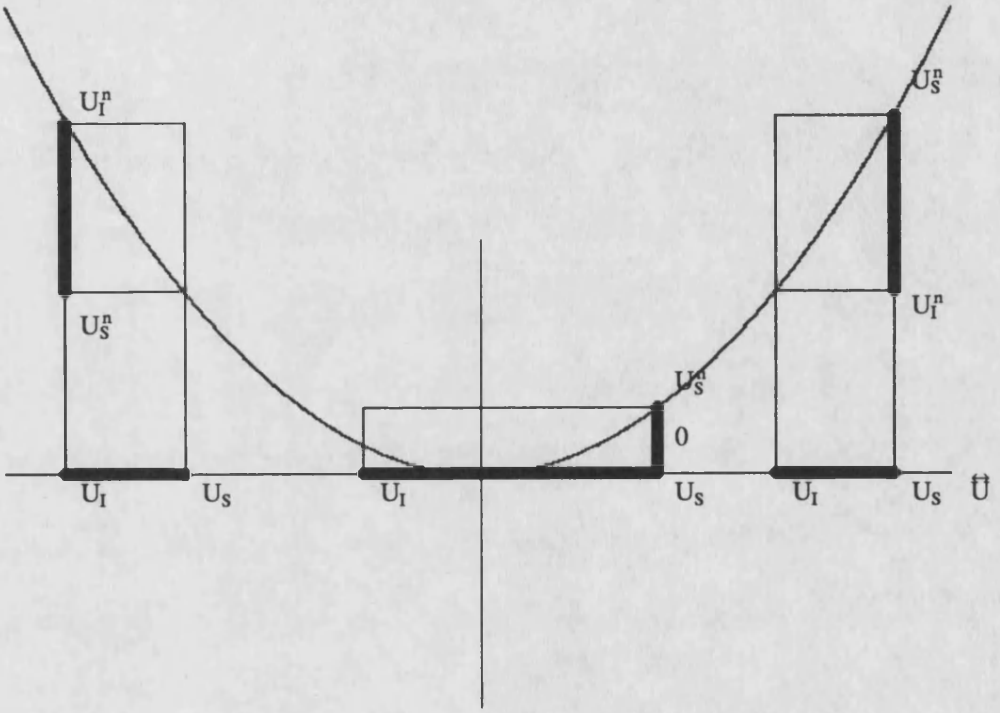


Fig 5.8.1a

Interval product : $\vec{U}\times\vec{V}$

Visualised as an enlargement map

$$0 < \vec{U} \rightarrow \begin{cases} 0 > \vec{V} \rightarrow \vec{U}\times\vec{V} = [U_I,U_S]\times[V_I,V_S] = [U_SV_I,U_IV_S] \\ 0 \in \vec{V} \rightarrow \vec{U}\times\vec{V} = [U_I,U_S]\times[V_I,V_S] = [U_SV_I,U_SV_S] \\ 0 < \vec{V} \rightarrow \vec{U}\times\vec{V} = [U_I,U_S]\times[V_I,V_S] = [U_IV_I,U_SV_S] \end{cases}$$

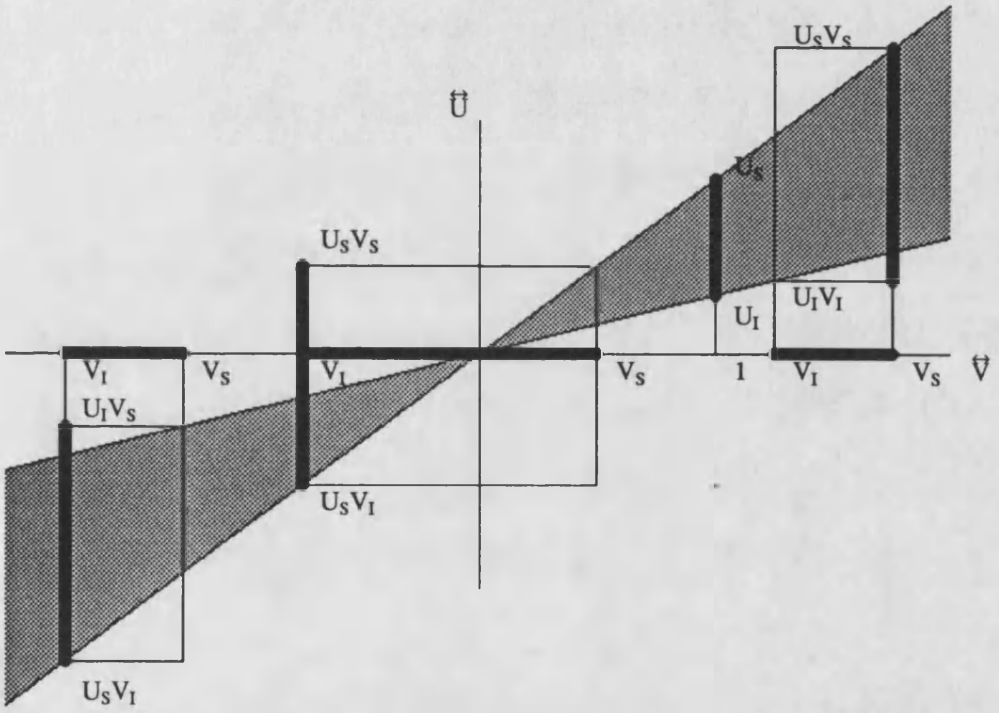


Fig 5.8.1a

Interval product : $\vec{U}\times\vec{V}$

Visualised as an enlargement map

$$0 \in \vec{U} \rightarrow \begin{cases} 0 > \vec{V} \rightarrow \vec{U}\times\vec{V} = [U_I,U_S]\times[V_I,V_S] = [U_SV_I,U_IV_I] \\ 0 \in \vec{V} \rightarrow \vec{U}\times\vec{V} = [U_I,U_S]\times[V_I,V_S] = [\min(U_SV_I,U_IV_S),\max(U_SV_S,U_IV_I)] \\ 0 < \vec{V} \rightarrow \vec{U}\times\vec{V} = [U_I,U_S]\times[V_I,V_S] = [U_IV_S,U_SV_S] \end{cases}$$

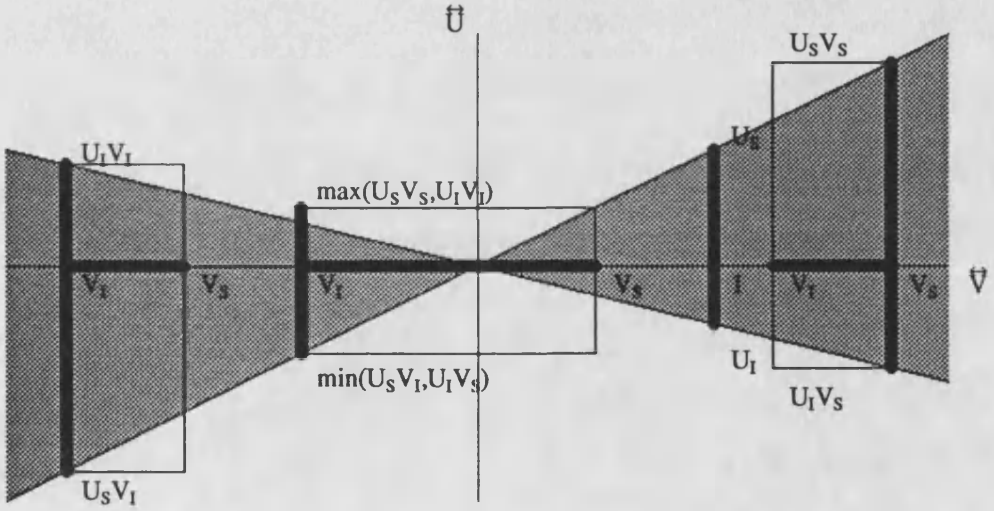


Fig 5.8.1a

Interval product : $\vec{U}\times\vec{V}$

Visualised as an enlargement map

$$0 > \vec{U} \rightarrow \begin{cases} 0 > \vec{V} \rightarrow \vec{U}\times\vec{V} = [U_I,U_S]\times[V_I,V_S] = [U_SV_S,U_IV_I] \\ 0 \in \vec{V} \rightarrow \vec{U}\times\vec{V} = [U_I,U_S]\times[V_I,V_S] = [U_IV_S,U_IV_I] \\ 0 < \vec{V} \rightarrow \vec{U}\times\vec{V} = [U_I,U_S]\times[V_I,V_S] = [U_IV_S,U_SV_I] \end{cases}$$

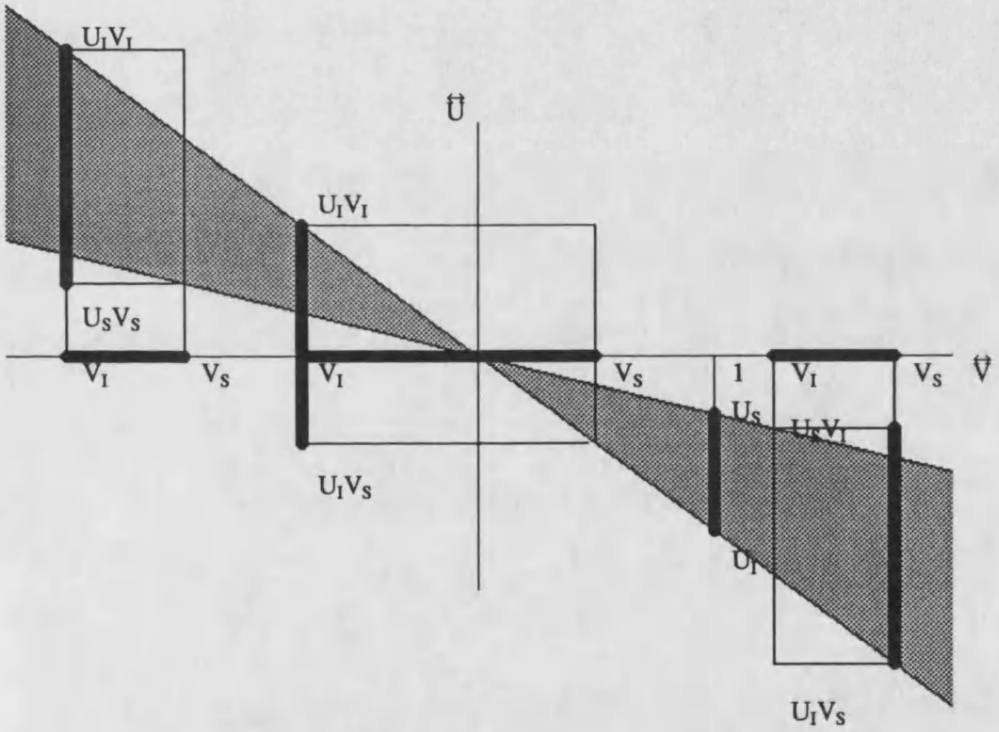


Fig 5.8.1a

Interval addition : $\vec{U}+\vec{V}$

Visualised as a vector graph

$$\vec{U}+\vec{V} = [U_I,U_S]+[V_I,V_S] = [U_I+V_I,U_S+V_S]$$

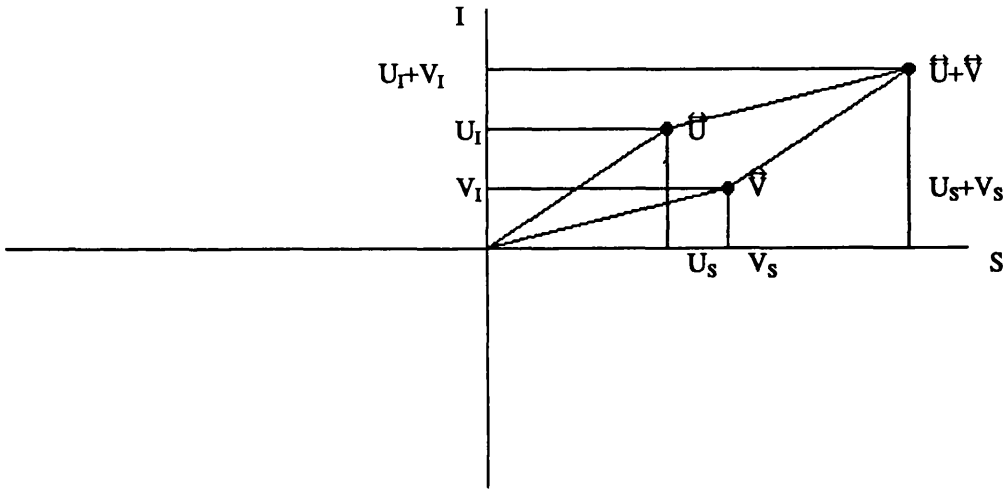


Fig 5.8.1a

Interval extrema : $\min(\vec{U},\vec{V})$, $\max(\vec{U},\vec{V})$

Visualised as a vector graph

$$\min(\vec{U},\vec{V}) = \min([U_I,U_S],[V_I,V_S]) = [\min(U_I,V_I),\min(U_S,V_S)]$$

$$\max(\vec{U},\vec{V}) = \max([U_I,U_S],[V_I,V_S]) = [\max(U_I,V_I),\max(U_S,V_S)]$$

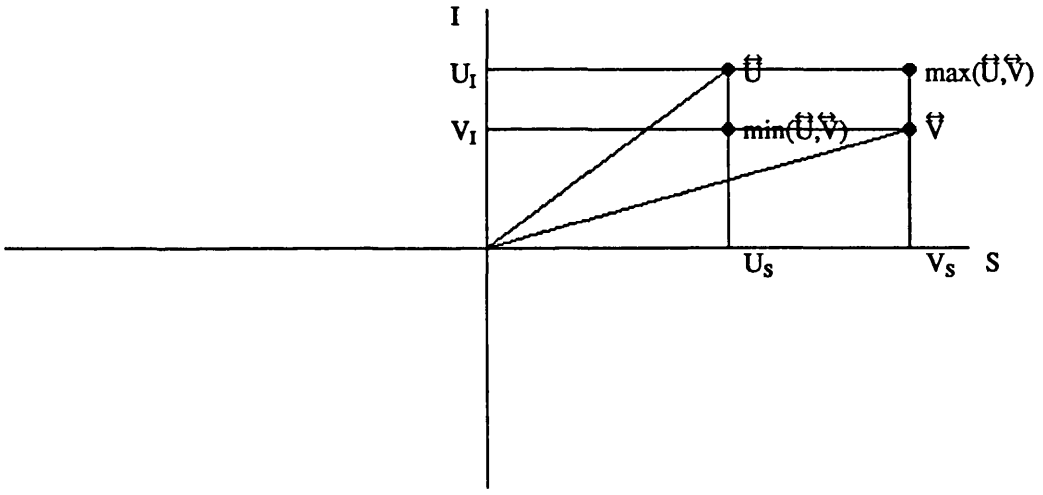


Fig 5.8.1a

Scalar operations

Any scalar λ may be considered as the degenerate interval $[\lambda,\lambda]$ of width zero. Scalar operations on intervals may then be defined in terms of the previous interval analysis. In particular,

Scalar interval multiplication : $\lambda\vec{U}$

$$\lambda\vec{U} = [\lambda,\lambda]\times[U_I,U_S] = \begin{cases} \lambda \geq 0 \rightarrow [\lambda U_I, \lambda U_S] \\ \lambda < 0 \rightarrow [\lambda U_S, \lambda U_I] \end{cases}$$

Interval negation : $-\vec{U}$

$$-\vec{U} = -1.\vec{U} = -1.[U_I,U_S] = [-U_S,-U_I]$$

Scalar interval addition $\lambda+\vec{U}$

$$\lambda+\vec{U} = [\lambda,\lambda]+[U_I,U_S] = [\lambda+U_I,\lambda+U_S]$$

Fig 5.8.1a

inside, outside or conservatively taken to contain part of the surface of the object if this height is uniformly negative, uniformly positive or contains zero respectively.

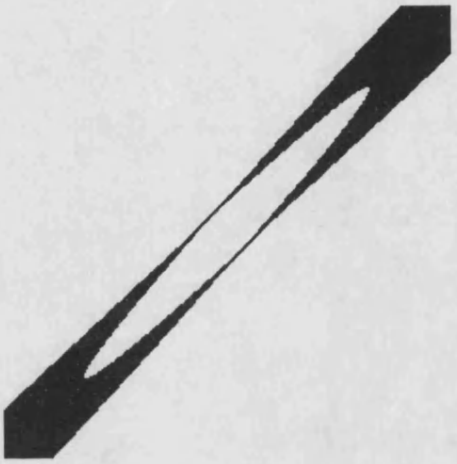
The evaluation of a height interval under univariate exponentiation and bivariate addition, multiplication, minimum and maximum necessitates a specification of height functions involving only these operations. When expressed in world coordinates, height functions generally include many dependent terms. The formal substitution of intervals corresponding to a world voxel into such an expansion can lead to catastrophic over-approximation of interval arithmetic in world space. The classification of a HOMO primitive's state often degenerates to the conservative HETERO case for voxels distant from the primitive's centre due to the high degree of term dependency [Fig 5.8.1b].

When expressed in local coordinates however, height functions generally consist of independent terms. Interval analysis is far more accurate when performed in local space, but cannot be directly applied to the local image of a given world voxel. Whilst the local image of a world point under linear transformation remains a point, world voxels become not local voxels but more general *parallelepipeds*. The local image of a voxel no longer constitutes independent intervals in each dimension. This is the domain over which interval arithmetic is defined. The local parallelepiped is conservatively approximated with a local voxel bound to allow interval arithmetic in this space.

The local parallelepiped image of a world voxel is easily determined for any linear transformation. It is simply the convex hull taken over the pointwise images of the world voxel's vertices. Such point images may be found with the vector and matrix arithmetic of usual linear algebra. Voxels may be taken as 3D vectors over intervals rather than points. Linear algebra may be defined for such vectors analogously to that for point vectors. Real linear algebra is generalised to interval linear algebra by simply replacing all real arithmetic with the equivalent interval arithmetic. Many of the usual associative and distributive laws carry over to this definition. The local voxel bound of a world voxel's parallelepiped image is then found by applying the transformation in this linear interval algebra [Fig 5.8.1c]. The formal substitution of the derived local voxel's intervals into a primitive's local height

Fig 5.8.1b: The Accuracy of Interval Analysis in World and Local space

Each diagram shows the cross-section of an octtree decomposition down to the surface of a cylinder by interval analysis. The shaded regions remain candidates to contain a section of the surface. The upper diagram illustrates the over-approximation caused by dependent terms in world space. The lower diagram shows the accuracy of decomposition in local space, avoiding such dependent terms.



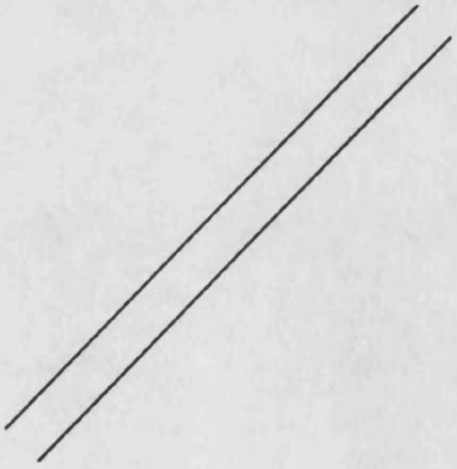


Fig 5.8.1b

Fig 5.8.1c: Interval Analysis of Linear Algebra

Notation

Symbol	Meaning
$\vec{R} = [R_I, R_S]$	Real interval between specified infimum R_I and supremum R_S
$\underline{p} = [p_x, p_y, p_z]$	Real vector
$V = [\vec{V}_x, \vec{V}_y, \vec{V}_z]$	Voxel or *interval vector.*
$L = [\underline{l}_1, \underline{l}_2, \underline{l}_3]$	World to local deformation matrix described as column vectors
$\underline{c}$	World shift component of world to local transform

Derivation

The image of a world voxel under linear transformation to local space is a parallelepiped. A voxel bound is calculated for this image to allow interval analysis in local space. Clearly, the appropriate local voxel giving the tightest possible fit may clearly be found by generalising the transformation's linear algebra from real to interval arithmetic. A definition is easily derived in terms of the previously given interval arithmetic [Fig 5.8.1a].

The Interval Scaling of a Real Vector to a Voxel

Let $V = \vec{R}_{\underline{p}}$ be the tightest voxel bound of the line segment resulting from the interval scaling of a vector. Then clearly

$$V = [\, \vec{R}p_x, \vec{R}p_y, \vec{R}p_z \,]$$

where the scaling of an interval by a real is defined as before [Fig 5.8.1a].

The Addition of Two Voxels to a Voxel

Let $W = U + V$ be the voxel of values assumed by the addition of two vectors where each ranges over a given voxel. Then clearly

$$W = [\, \vec{U}_x+\vec{V}_x, \vec{U}_y+\vec{V}_y, \vec{U}_z+\vec{V}_z]$$

where the addition of intervals is defined as before [Fig 5.8.1a].

The Multiplication of a Voxel by a Real Matrix to a Voxel

Let $U = LV$ be the tightest voxel bound of the parallelepiped resulting from the multiplication of a voxel by a real matrix. Then clearly

$$U = \vec{U}_x\underline{l}_1 + \vec{U}_y\underline{l}_2 + \vec{U}_z\underline{l}_3$$

where the interval scaling of a real vector to a voxel and the addition of voxels is defined as above.

The Vector Shift of a Voxel to a Voxel

Any vector may be taken as a degenerate voxel with no spread. The vector may then be taken to shift a voxel according to the addition of voxels given above.

The Linear Transformation of a Voxel from World to Local Space

Let the linear transformation consist of a vector shift and matrix deformation giving the point wise mapping $\underline{p} \rightarrow L(\underline{p}-\underline{c})$. Let U be the tightest local voxel bound of a world voxel's parallelepiped image under this transformation. Then clearly

$$U = L(V - \underline{c}) = LV - L\underline{c}$$

where the vector shift and real matrix multiplication of a voxel are defined as above.

function provides a closer state approximation than over a world voxel. Interval arithmetic is easily derived for the state classification of any primitive or construct [Fig 5.8.1d].

Over-approximation may still occur however. Transformations to a local space where a voxel is a poor approximation of the world voxel's true parallelepiped image are particularly vulnerable. A typical case is a transformation comprising a stretch along a world axis followed by rotation through a non-right angle. Excessive approximation results from dependencies within the local parallelepiped being ignored over the local voxel bound. Customised *trivariate* interval arithmetic may be defined on voxels to recognise these dependencies in specific cases. This can provide a closer approximation of a primitive's height interval over a world voxel.

5.8.2. Interval Analysis of Trivariate Arithmetic

The reader may have noticed that the *square of a local point's modulus* is a particularly common term in the local height functions of the modelled primitives [APPENDIX B]. Whilst the interval assumed by this term over a local parallelepiped may be conservatively approximated from that of a local voxel bound, a voxel of poor fit can result in a poor approximation. Trivariate analysis provides a closer approximation of this term's interval. Consider the interval of the square of local modulus taken over the parallelepiped image of a world voxel under a given linear transform. This interval is clearly non-negative, and will always attain a maximum at a parallelepiped vertex. For the special case of a parallelepiped containing the local origin it will assume a minimum of zero.

This observation may be exploited by considering world voxels which transform to local parallelepipeds centred at the local origin. A voxel may be specified by two vectors. A *centre* vector specifies its location in each dimension, and a *radius* vector specifies its spread. The voxel is then defined as the sum of its centre vector and a world origin centred *residue* voxel of equal radius.

Consider the transformation of a world voxel to local space. This constitutes a vector shift followed by a matrix deformation. The shift may be restricted to the voxel centre without

Fig 5.8.1d: Interval Analysis for Conservative State Code Approximation

Notation

Symbol	Meaning
$L = [\underline{l}_1,\underline{l}_2,\underline{l}_3]$	World to local deformation matrix described as column vectors
$\underline{c}$	World centre of primitive instance
$\underline{p}$	World point
$\underline{l} = L(\underline{p}-\underline{c})$	Local image of world point
V	World voxel
$U = L(V-\underline{c})$	Local voxel bound of world voxel's parallelepiped image under linear transformation
A, B	Boolean constructs
h_A	Real height above construct A
$\tilde{H}_A$	Conservative approximation of height interval above construct A

Derivation

Interval analysis is applied to conservatively approximate the interval of each primitive's local height function over the local voxel bound of a world voxel's parallelepiped image. If this height interval is uniformly positive, the world voxel is known to be entirely outside the primitive instance. If it is uniformly negative, the world voxel is known to be entirely inside the primitive instance. Otherwise the interval contains zero and the world voxel is conservatively taken to contain a section of the primitive instance's surface.

This analysis is generalised from primitives to boolean constructs. The interval arithmetic is defined as before [Fig 5.8.1a]. In each case it is a direct analogy of the real arithmetic to test whether a point is within an object [APPENDIX B].

The Plane

Real Arithmetic [APPENDIX B]:

$$h_{Plane} = l_y$$

Interval analogy:

$$\overleftrightarrow{H}_{Plane} = \overleftrightarrow{U}_y$$

The Cube of X extent $[X_{min},X_{max}]$, Y extent $[Y_{min},Y_{max}]$, Z extent $[Z_{min},Z_{max}]$

Real Arithmetic [APPENDIX B]:

$$h_{Cube} = \max \left\{ \begin{array}{l} \max\{l_x - X_{max},\ X_{min} - l_x\}, \\ \max\{l_y - Y_{max},\ Y_{min} - l_y\}, \\ \max\{l_z - Z_{max},\ Z_{min} - l_z\} \end{array} \right\}$$

Interval Analogy:

$$\overleftrightarrow{H}_{Cube} = \max \left\{ \begin{array}{l} \max\{\overleftrightarrow{U}_x - X_{max},\ X_{min} - \overleftrightarrow{U}_x\}, \\ \max\{\overleftrightarrow{U}_y - Y_{max},\ Y_{min} - \overleftrightarrow{U}_y\}, \\ \max\{\overleftrightarrow{U}_z - Z_{max},\ Z_{min} - \overleftrightarrow{U}_z\} \end{array} \right\}$$

The Sphere of Radius R

Real Arithmetic [APPENDIX B]:

$$h_{Sphere} = l_x^2 + l_y^2 + l_z^2 - R^2$$

Interval Analogy:

$$\overleftrightarrow{H}_{Sphere} = \overleftrightarrow{U}_x^2 + \overleftrightarrow{U}_y^2 + \overleftrightarrow{U}_z^2 - R^2$$

The Cylinder of Radius R, Y extent $[Y_{min},Y_{max}]$

Real Arithmetic [APPENDIX B]:

$$h_{Cylinder} = \max \left\{ \begin{array}{c} l_x^2+l_z^2-R^2, \\ \max\{l_y-Y_{max},\ Y_{min}-l_y\} \end{array} \right\}$$

Interval Analogy:

$$\overleftrightarrow{H}_{Cylinder} = \max \left\{ \begin{array}{c} \overleftrightarrow{U}_x^2+\overleftrightarrow{U}_z^2-R^2, \\ \max\{\overleftrightarrow{U}_y-Y_{max},\ Y_{min}-\overleftrightarrow{U}_y\} \end{array} \right\}$$

The Double Cone of Axial Angle α, Y extent $[Y_{min},Y_{max}]$

Real Arithmetic [APPENDIX B]:

$$h_{Cone} = \max \left\{ \begin{array}{c} (l_x^2+l_z^2)\cos^2(\alpha)-l_y^2\sin^2(\alpha), \\ \max\{\ l_y - Y_{max},\ Y_{min} - l_y\} \end{array} \right\}$$

Interval Analogy:

$$\overleftrightarrow{H}_{Cone} = \max \left\{ \begin{array}{c} (\overleftrightarrow{U}_x^2+\overleftrightarrow{U}_z^2)\cos^2(\alpha)-\overleftrightarrow{U}_y^2\sin^2(\alpha), \\ \max\{\ \overleftrightarrow{U}_y - Y_{max},\ Y_{min} - \overleftrightarrow{U}_y\} \end{array} \right\}$$

The Torus of Major Axis R, Minor Axis r

Real Arithmetic [APPENDIX B]:

$$h_{Torus} = \left[l_x^2+l_y^2+l_z^2-(R^2+r^2)\right]^2 + 4R^2(l_y^2-r^2)$$

Interval Analogy:

$$\overleftrightarrow{H}_{Torus} = \left[\overleftrightarrow{U}_x^2+\overleftrightarrow{U}_y^2+\overleftrightarrow{U}_z^2-(R^2+r^2)\right]^2 + 4R^2(\overleftrightarrow{U}_y^2-r^2)$$

Cut-Aways of Decompositions to the

Surfaces of Various Primitives

Using Interval Analysis

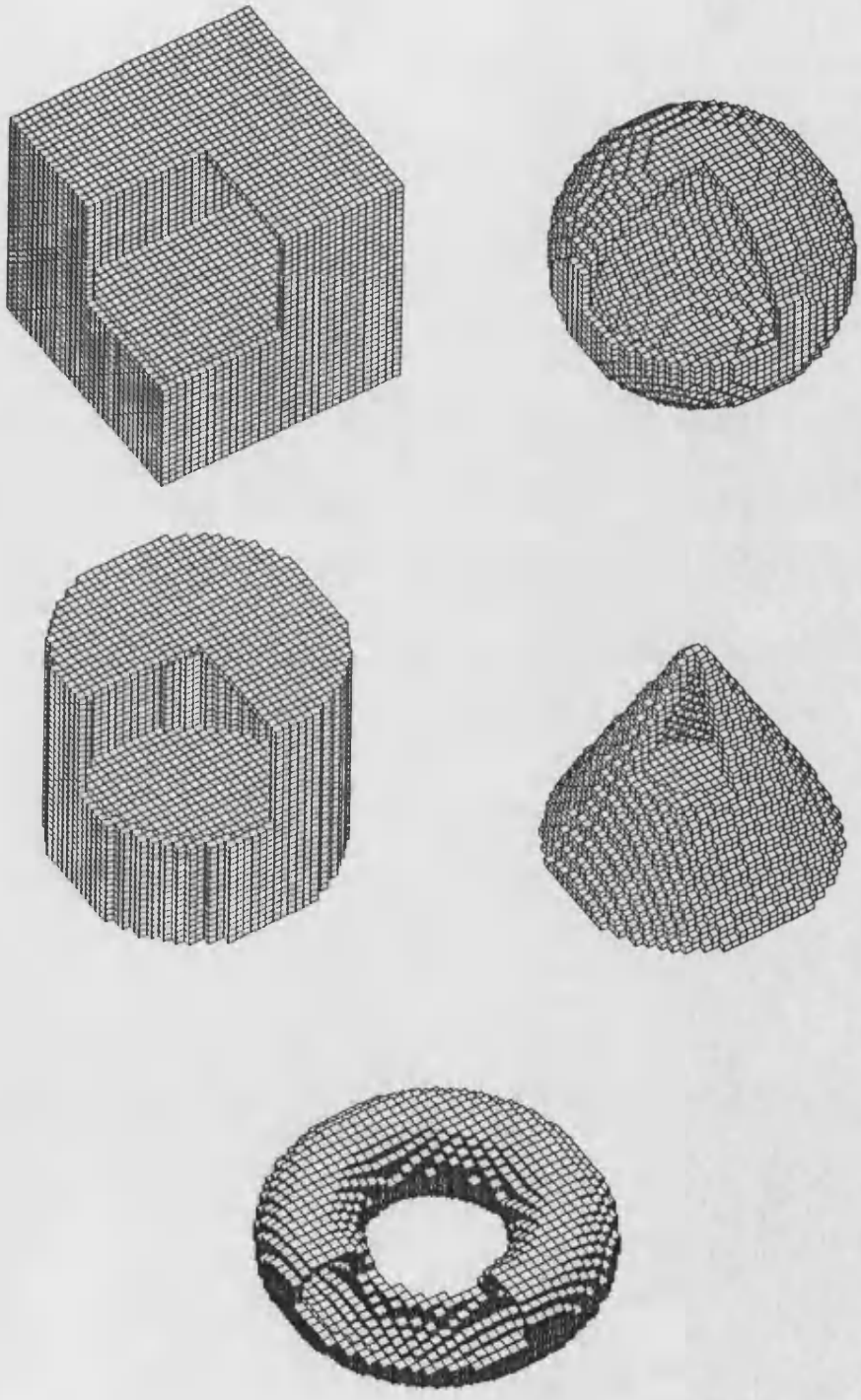


Fig 5.8.1d

Boolean Operations

Real Arithmetic [APPENDIX B]:

$$h_{A \text{ union } B} = \min(h_A, h_B)$$

Interval Analogy:

$$\vec{H}_{A \text{ union } B} = \min(\vec{H}_A, \vec{H}_B)$$

In a heterogeneity test, $\vec{H}_A < 0 \rightarrow \min(\vec{H}_A, \vec{H}_B) = \vec{H}_{A \text{ union } B} < 0$ with no need to actually calculate $\vec{H}_B$

Real Arithmetic [APPENDIX B]:

$$h_{A \text{ intersect } B} = \max(h_A, h_B)$$

Interval Analogy:

$$\vec{H}_{A \text{ intersect } B} = \max(\vec{H}_A, \vec{H}_B)$$

In a heterogeneity test, $\vec{H}_A > 0 \rightarrow \max(\vec{H}_A, \vec{H}_B) = \vec{H}_{A \text{ intersect } B} > 0$ with no need to actually calculate $\vec{H}_B$

Real Arithmetic [APPENDIX B]:

$$h_{A \text{ subtract } B} = \max(h_A, -h_B)$$

Interval Analogy:

$$\vec{H}_{A \text{ subtract } B} = \max(\vec{H}_A, -\vec{H}_B)$$

In a heterogeneity test, $\vec{H}_A > 0 \rightarrow \max(\vec{H}_A, -\vec{H}_B) = \vec{H}_{A \text{ subtract } B} > 0$ with no need to actually calculate $\vec{H}_B$

Real Arithmetic [APPENDIX B]:

$$h_{A \text{ difference } B} = \max(\min(h_A, h_B), -\max(h_A, h_B))$$

Interval Analogy:

$$\vec{H}_{A \text{ difference } B} = \max(\min(\vec{H}_A, \vec{H}_B), -\max(\vec{H}_A, \vec{H}_B))$$

Cut-Aways of Decompositions to the

Surfaces of Various Sphere CSG Constructs

Using Interval Analysis

Top Left: Union Top Right: Intersection

Bottom Left: Subtraction Bottom Right: Symmetric Difference

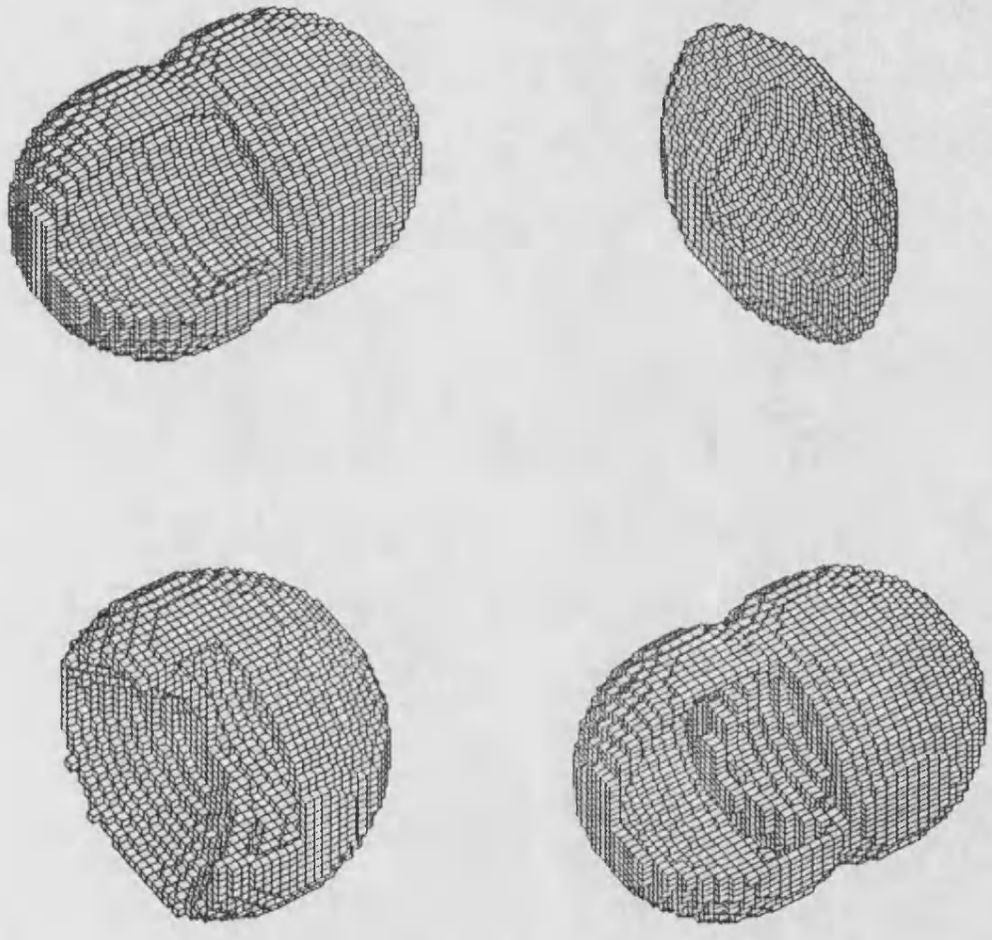


Fig 5.8.1d

translating its residue. By linearity, the matrix deformation may then be applied to the shifted centre separately from the voxel residue. The former is transformed to a local centre in the usual pointwise fashion. However, the world residue voxel is transformed to a local parallelepiped centred at the local origin.

The square of local modulus may be expressed as the auto dot product of the local vector centre and parallelepiped's sum. This expands to the sum of three terms: the real auto dot product of the local centre; twice the interval over the parallelepiped of the dot product with the local centre ; and the interval over the parallelepiped of local modulus squared . The first can be evaluated exactly with real linear algebra, whilst the others can be found with trivariate interval analysis directly from the parallelepiped vertices *without* resorting to a local voxel approximation.

Whilst this arithmetic is well defined it incurs the calculation of candidate maxima at each of eight vertices. The appropriate vertex may sometimes be identified without an exhaustive search, but dot product arithmetic is still required to calculate the extreme assumed there. Fortunately however the octtree decomposition deals not with arbitrary voxels but a regular voxel hierarchy. A simple recurrence relation holds for the maximum of the square of local modulus over the origin centred parallelepipeds corresponding to successive octtree generations. This relation is division by four, generalising to a recurrence relation over 'n' generations of division by 4^n. The minimum remains zero throughout. The relatively costly trivariate arithmetic need only be performed once per primitive for the scene's root voxel. Appropriate intervals are then found at any level of decomposition through division by the relevant factor. This factor is propagated down recursion through multiplication by four. Where applicable, the state of an object classified as HOMO by univariate and bivariate analysis over a local parallelepiped's voxel approximation may be reclassified with this trivariate analysis directly over the parallelepiped.

5.8.3. The Increasing Accuracy of Interval Analysis During Octtree Decomposition

The error of conservative approximation by interval arithmetic in local space decreases with voxel size. Each coordinate in a local voxel bound where a height function's term assumes

an extreme may be expressed as the sum of the parallelepiped coordinate where the entire function assumes that extreme and an error term. Such expressions may be substituted back into the height function, yielding a polynomial which may be expanded by the binomial theorem. This may be rearranged to express the approximation as the sum of the true extreme and a polynomial error expression. Every term of the error expression contains at least one coordinate error factor, each of which is bounded by local voxel width. The degree of approximation therefore decreases with voxel size, with exact classification in the decomposition limit.

5.9. Drawbacks of the Grid Partition

A scene grid partition into uniform voxel cells is the immediate 3D generalisation of a 2D raster pixel array. This is a conceptually simple structure and efficient navigation algorithms are easily generalised from other well understood applications.

However, the grid partition does not adapt to local scene complexity. A typical scene will comprise large coherent regions in which every object is homogeneous. Such regions cannot be made any simpler. Their decomposition provides no gain but rather produces an excessive number of grid cells. This increases both storage requirements and the average number of voxels navigated by a ray and is therefore undesirable. Homogeneous regions should ideally be left unpartitioned, motivating a grid of low resolution. The simplification of heterogeneous regions to an acceptable level however motivates a grid of high resolution. The goal of an efficient decomposition over a homogeneous region is clearly at odds with that over a heterogeneous region. The uniform grid partition is too rigid to meet these twin needs.

The octtree decomposition has been shown to be an efficient means of generating grid partitions. This quickly identifies large homogeneous regions. The octtree decomposition ignores such regions to focus attention solely on heterogeneous regions. This adaptive decomposition is well suited to the simplification of any scene region. Several researchers have proposed the acceleration of ray tracing by the direct navigation of the octtree itself [Fujimoto et al;1986: Glassner;1984,1988: Haines;1988]. Several octtree navigation

algorithms have been proposed, but have tended to be inefficient when compared to the navigation of grid partitions [Fujimoto et al;1986: Glassner;1984,1988]. As the 3D generalisation of the quadtree, the octtree is perhaps a less familiar structure than the grid partition. Many researchers have favoured the grid partition whose navigation and construction have been better understood. They do not exploit the advantages of the octtree due to the inefficiency of navigation algorithms to date. Chapter six of this thesis derives an efficient ray navigation algorithm for octtrees, overcoming this obstacle to allow their full exploitation.

Chapter 6: Octtree Decompositions

Synopsis:

Chapter six addresses the decomposition of a scene by an octtree. Both the use and generation of this decomposition are considered. A particularly efficient ray navigation algorithm is derived which allows the full exploitation of the octtree.

6.1. The Simplification of the Scene Model with Octtree Scene Decompositions

The octtree decomposition facilitates the scene model's solution in a similar manner to the grid partition [Section 5.1]. A box enclosing the scene is decomposed into simpler voxels [Fig 6.1a]. A given ray's solution to the scene model first queries this box for ray intersection. If missed none of the scene is significant and the model is solved trivially. Otherwise, the ray entrance point is found and transformed from the world to a local coordinate system in which the box is a unit cube. The ray is navigated through the leaf voxels of the octtree decomposition in path length order [Fig 6.1b; 6.1c]. The objects from each leaf voxel's heterogeneous list are queried in turn for the scene model's solution. Repeated object query is avoided as before [Section 5.1.1]. Each voxel's heterogeneous list is reduced from the total scene count so that costs in object queries are traded for the ray's navigation between leaf voxels. This navigation should be as efficient as possible to maximise any computational savings.

6.2. Previous Octtree Representations and Associated Ray Navigation Algorithms

Various algorithms for ray navigation through an octtree have been published [Glassner;1984: Fujimoto et al;1986: Haines;1988]. However, these have tended to be rather inefficient.

6.2.1. Direct Navigation Between Leaf Voxels under Floating Point Arithmetic

Glassner [1984] published an early paper on simplifying the solution of the scene model with octtree scene decompositions. This proposed using Gargantini's leaf code [Gargantini;1982] to store only an octtree's leaf voxels without the internal node structure. Rays were navigated in direct leaps between these leaves. Each leap comprised several steps within the associated tree diagram. The ray's exit point from the current leaf voxel was found by intersection with all six clipping planes in multiplicative floating point arithmetic. Glassner claimed that only four of these intersections were significant but the effort to identify which four outweighed the advantages of eliminating two intersections. This claim is rather surprising since only three intersections need actually be considered.

Fig 6.1a: Non-Empty Voxels in Cut-Aways
of Alternative Octtree Decompositions for a Given Scene

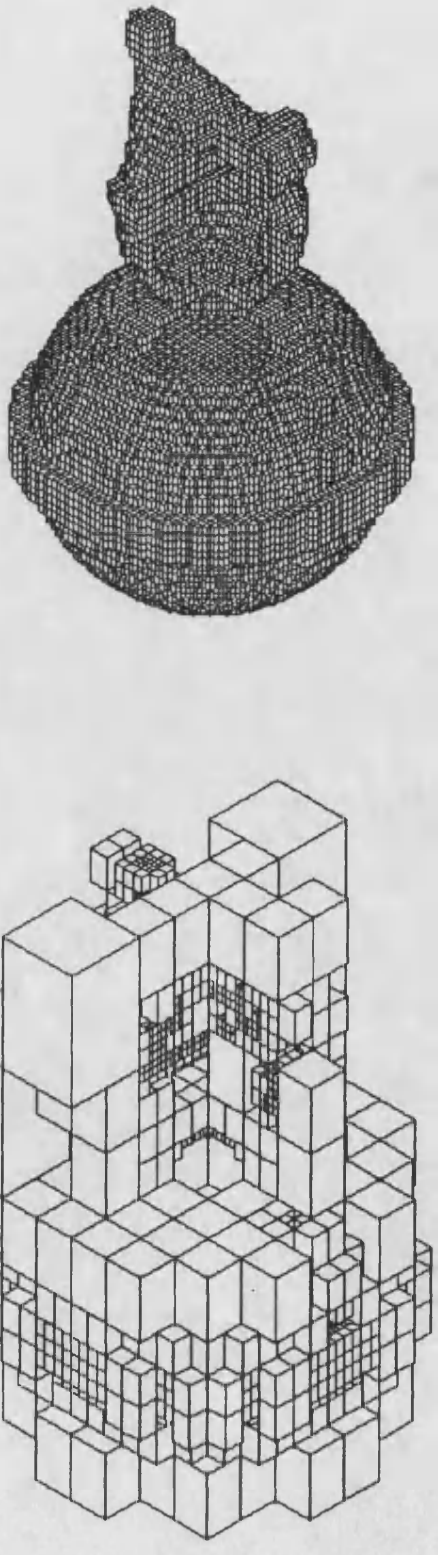
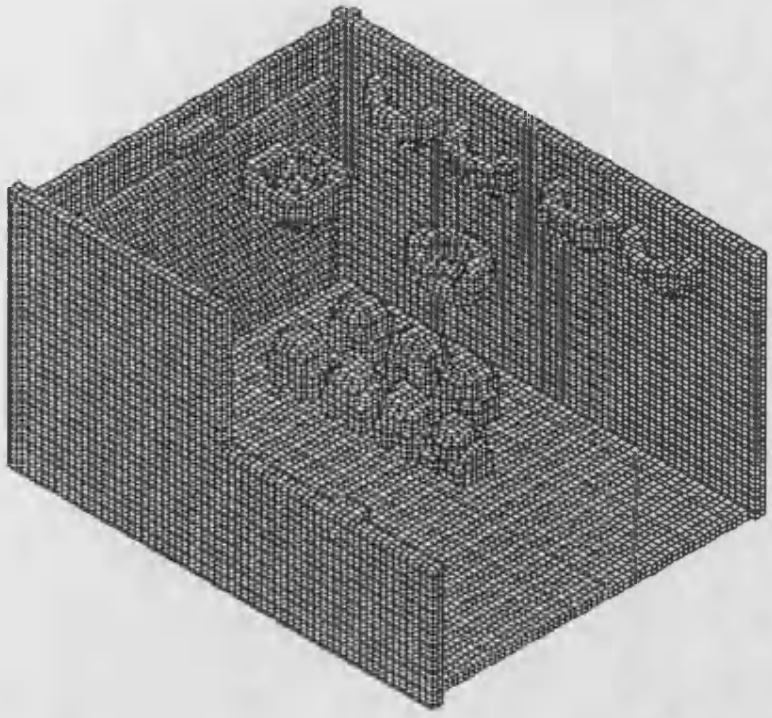


Fig 6.1a



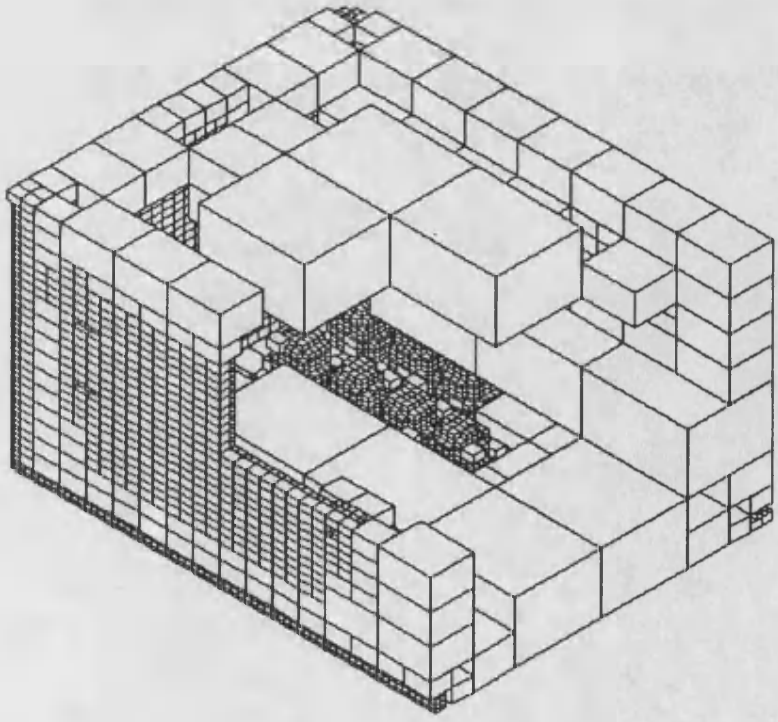


Fig 6.1a

Fig 6.1b: The 2D Ray Navigation of a Quadtree

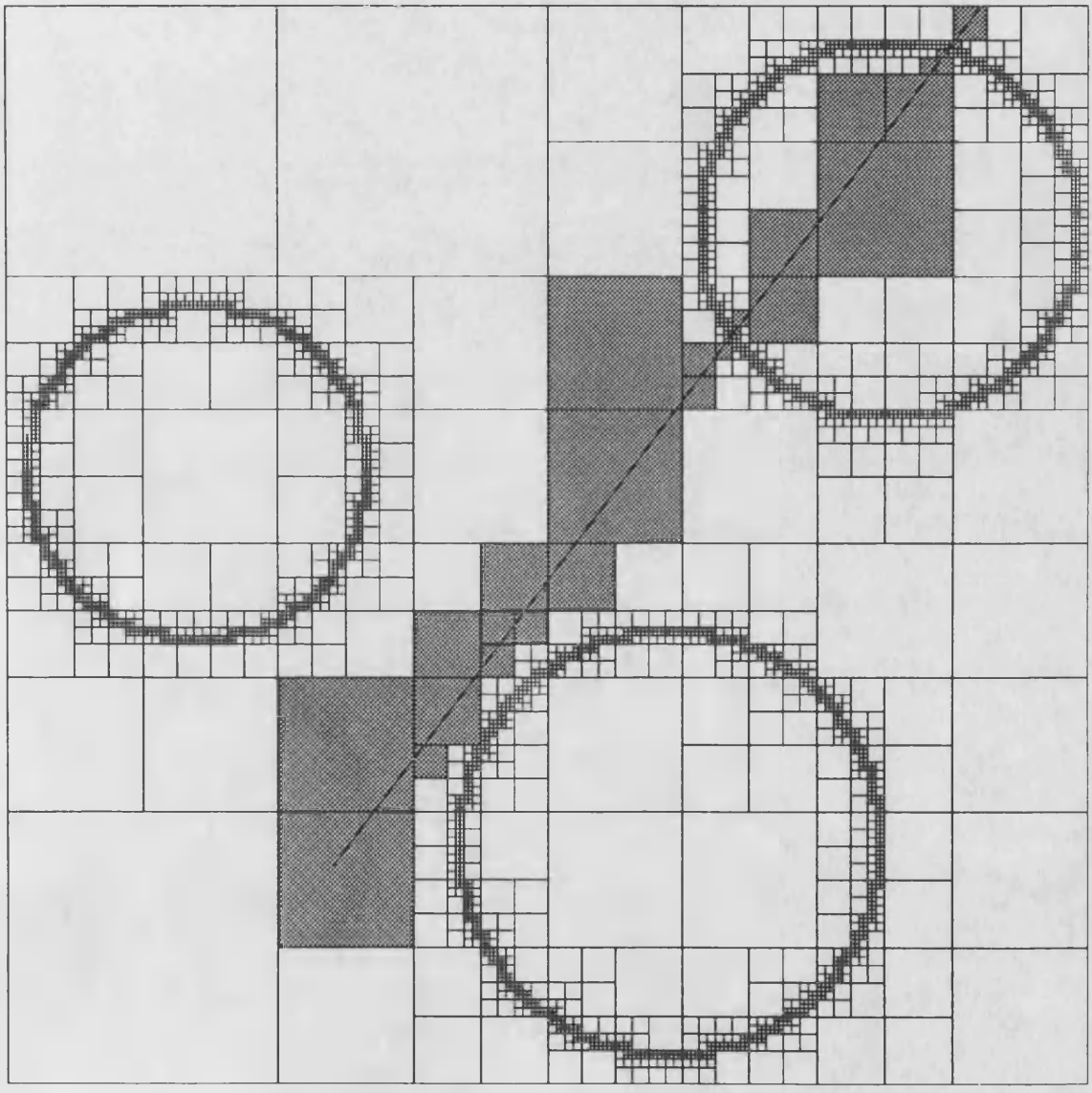


Fig 6.1b

Fig 6.1c: The 3D Ray Navigation of an Octtree

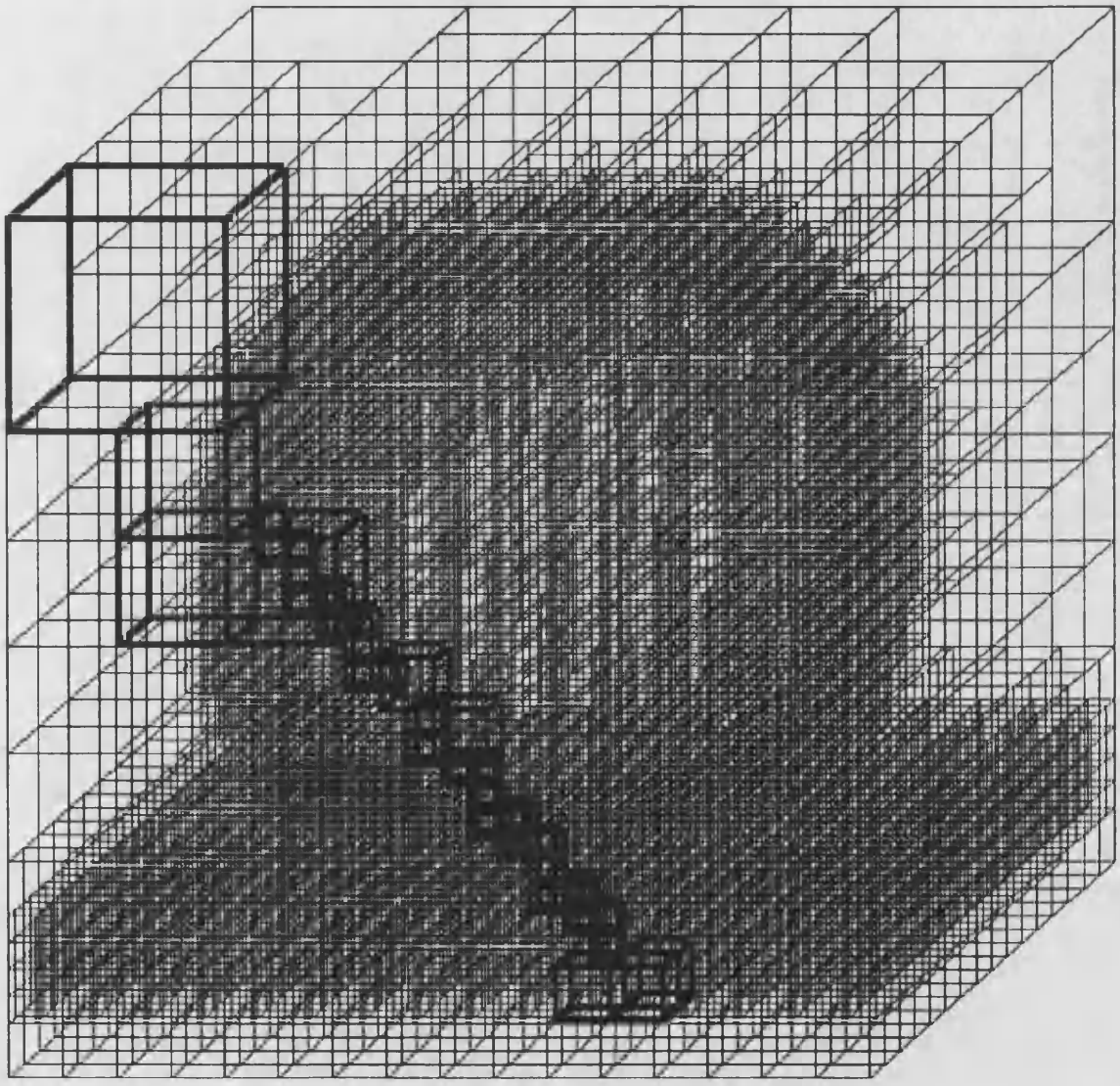


Fig 6.1c

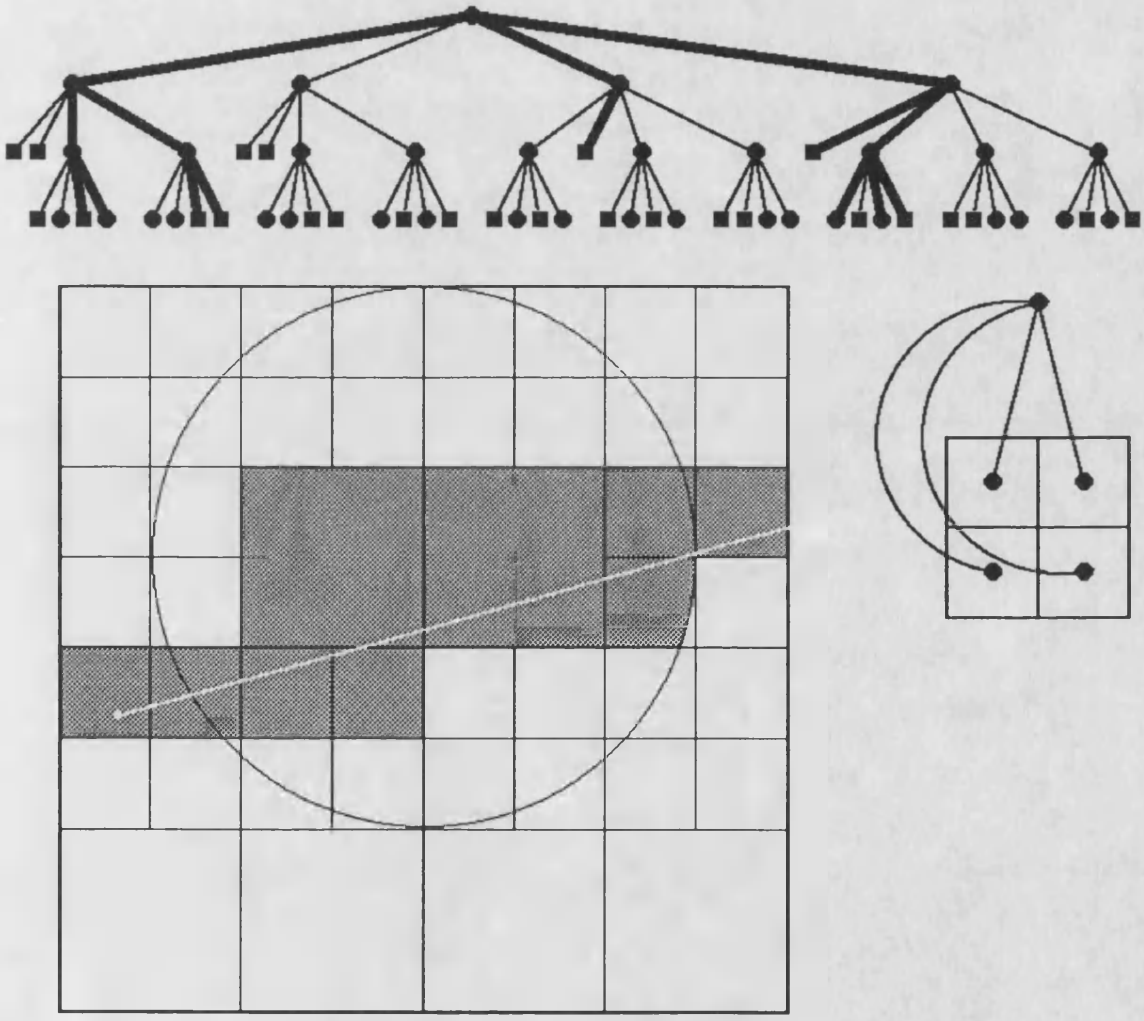
These are always those planes in the same octant relative to the voxel centre as the ray direction. Moreover, since direction is constant for any given ray the set of significant planes may be encoded once per ray in a single octant digit without recalculation over every step. The Morton coding scheme is particularly convenient and requires little overhead [Figure 5.6.2a]. Each step between leaf nodes was navigated to the neighbouring voxel which shared this exit point. The post-step voxel was located via an internal point at a given offset from the shared face point. The leaf voxel containing this point was located through a hash table with an implicit return to the octtree root. Whilst Glassner's navigation achieved a faster image synthesis than naive ray tracing, the twin overheads of multiplicative floating point arithmetic for navigation and hash table look up to locate the post-step voxel are clearly counter productive.

6.2.2. Navigation by Single Octtree Steps under Incremental Fixed Point Arithmetic

An incremental algorithm employing a 3D DDA would require only additive arithmetic to navigate each step. An enhancement similar to Bresenham's algorithm could multiply out all division from the increment expressions. This would avoid degenerate exceptions. It would also uniformly bound all decision variables to minimise numerical inaccuracies in their quantisation for maintenance under fixed point arithmetic. The DDA relies on taking constant steps of unit cell width along the driving axis within a grid partition. Movement by one cell along the driving axis is guaranteed since each is of uniform width. This would be a reckless step to attempt between the leaf voxels of an octtree however, since these may differ in size. A constant width step along the driving axis would produce a movement over several neighbouring leaves of smaller width yet may not escape a leaf of greater width.

The Accelerated Ray Tracing System ARTS [Fujimoto et al;1986] overcame this obstacle by dividing each step between leaf voxels into a sequence of single steps within the octtree diagram. Two types of step were distinguished. These were a *vertical* step between voxels of different generations sharing the same ray point and a *horizontal* step between sibling voxels to a new ray point [Fig 6.2.2a]. Any post-step voxel was located directly from the pre-step voxel. There was no need to return to and search down from the root.

Fig 6.2.2a: The Navigation of a Tree Decomposition by a Sequence of Vertical and Horizontal Steps in the Tree Diagram



Each horizontal step is between siblings voxels of the same size. ARTS exploited this to navigate horizontal steps with a 3D DDA. Whilst vertical steps are between voxels of different generations, their size differs by a linear factor of two. ARTS exploited this to navigate vertical steps with an algorithm which scaled by-products of the 3D DDA by a factor of two. This is a particularly simple operation in binary computation. However, the details of ARTS' vertical navigation algorithm have not been published at the time of writing this thesis.

ARTS stored octtrees in a format which has been more fully described as *Autumnal* for the 2D quadtree [Fabbrinni,Montani;1986]. With the exception of a degenerate single node tree, the Autumnal format stores only internal nodes. This compact format occupies just under one eighth the space of the regular octtree representation. Moreover, sibling location is more efficient in Autumnal format. This facilitates the location of the next voxel after a horizontal step. An efficient location is most important in ray tracing where horizontal steps occur in ray dependent order between at most half the siblings, skipping over other entire sibling branches. Each internal octtree node is represented by an array of eight pairs, one per child node in Morton order [Figure 5.6.2a]. Each pair comprises a status code and pointer. Fabbrinni [1986] distinguished between three cases of status code for his 2D Autumnal quadtrees. The status code was conveniently combined with the paired pointer in a signed single field record. ARTS distinguished between four status codes however and represented each pair as an explicit two field record. These codes defined the status of the scene objects with respect to the associated voxel and differentiated between internal and leaf nodes. The codes were HOMO_OUT for a leaf child homogeneously outside every object, HOMO_IN for a leaf child homogeneously inside an object, HETERO_LEAF for a leaf child heterogeneous with respect to an object and HETERO_BRANCH for a similarly heterogeneous internal child. The status code defined the list indexed by the paired pointer to recursively identify the child's heterogeneous object list. HETERO_LEAF nodes referenced a global heterogeneous object list similar to that described for grid partitions [Section 5.5]. HETERO_BRANCH nodes referenced another internal node in the Autumnal

list. The list indexed by a child in either HOMO state was undefined since such children have no heterogeneous objects.

ARTS chose to order these internal nodes breadth first through memory. However, the Autumnal representation may be listed in any order convenient to octtree generation. An octtree is particularly easy to preprocess in depth first order before image synthesis, whilst dynamic generation during ray tracing may lead to an arbitrary fragmentation.

6.2.3. The Efficiency of Previous Octtree Navigation

Horizontal octtree steps were navigated in ARTS with the same 3D DDA as for a unit driving axis step in a partition grid [Section 5.2.3]. This did not exploit Bresenham's enhancement so neither avoided exceptions nor bounded decision variables uniformly. Experiments have been conducted on ARTS to compare the speed of image synthesis when navigating the octtree to that with the grid partition. It was "*expected that the octtree encoding would have an advantage for cases [of] high scene coherency. This will result in a smaller number of cells .... This means that ... a ray can reach the surface of an object by traversing fewer cells*". However, image synthesis was actually found to be slower when navigating an octtree than a grid partition. Since horizontal steps were navigated identically in each case, the conclusion was that whilst the average step count of navigation may have been lower for an octtree, "*this was outpaced by the cost of [their] vertical traversing of the octtree*". It was remarked that a costly vertical navigation could have a significant impact on rendering times since "*vertical traversing must be performed after at most four cells are identified ... during horizontal traversing [and] it may be needed as soon as one horizontal step is performed. Depending on the depth of the octtree, it may quite often be necessary to perform several steps of ascending and descending the octtree*". Any inefficiencies in ARTS' vertical step navigation cannot be pinpointed in the absence of a detailed description of the algorithm. Since this was omitted from the paper "*due to space limitations*", the algorithm would appear to be somewhat complicated. The complication presumably arises from the horizontal navigation of octtrees in sibling-width steps along the driving axis. If the post-step sibling were not a leaf, the current ray point would have to be backtracked along

the ray path to first descendent visited. This would be difficult to achieve in sibling-width steps or binary fractions thereof along the driving axis.

Whilst ARTS' navigation of an octtree was less efficient than that of a grid partition, it still proved more than thirteen times faster than Glassner's octtree navigation. This indicates the impact of navigation under fixed rather than floating-point arithmetic and simpler schemes for locating the post step voxel.

ARTS' navigated vertical steps iteratively. During descent, pointers were recorded in a short working array to be recalled during subsequent ascent. The navigation of both ascending and descending vertical steps required some arithmetic. Octtrees may be recursively navigated top down. Ascending vertical steps then always return to the known parent. The completion of a descending vertical step's recursion automatically takes a corresponding ascending vertical step back up the octtree with no need for navigational arithmetic. ARTS' navigation was implemented in non-recursive FORTRAN 77 and did not exploit this opportunity to reduce the vertical navigation count.

6.3. The SMART Navigation of an Octtree

The above motivates research for an octtree navigation which is more efficient over vertical steps, preferably still maintained in fixed point arithmetic. All decision variables should be uniformly bounded to minimise numerical errors in quantisation. This section of the thesis derives such an algorithm.

The octtree is navigated by a sequence of single vertical and horizontal steps within the octtree diagram as in ARTS. Any post-step voxel is located directly from the pre-step voxel without returning to the octtree root. Each step is navigated with a decision vector called a Spatial Measure for Accelerated Ray Tracing or SMART. A pair of SMARTs is maintained, one navigating vertical steps and the other horizontal steps. As in ARTS, the SMART navigation is maintained with integer logic and predetermined increments. Unlike ARTS however, these increments are not constant but adapt dynamically to each visited voxel. The mathematics of the navigation ensure that the appropriate increments are always

available in the current recursive environment, and so need not be recalculated over every step. Horizontal steps are not navigated in unit increments along a driving axis. Each horizontal step only goes as far as the ray exit point from the current sibling, much as in Glassner's navigation [Glassner;1984]. Rash steps which would overshoot several descendant leaves in a post-step sibling are avoided. Navigation within the sibling is straightforward since backtracking is no longer required. In layman's terms, the SMART navigation pays heed to the adage "*look before you leap*". Problems in overshooting render the driving axis concept rather artificial when applied to an octtree. The SMART octtree navigation finally abandons this somewhat anachronistic hangover from the navigation of grid partitions.

The SMART navigation theory is somewhat detailed [Fig 6.3a] but leads to a remarkably simple implementation. Whilst a textual English description of the algorithm's derivation is somewhat verbose, the program code is concise [Fig 6.3b].

The ray is only navigated between internal nodes. On reaching a leaf node, the associated heterogeneous objects are queried in turn for the ray's scene model solution. Repeated object query over leaf voxels is avoided as before [Section 5.1.1].

6.4. The Automatic Generation of an Octtree Scene Decomposition

6.4.1. Specifying the Octtree to be Generated

The essential structure of an octtree is predetermined much as for a grid partition. An octtree recursively decomposes a voxel into eight children until one of two termination criteria are met. The *simplicity* criterion checks that the length of the voxel's heterogeneous object list does not exceed some upper bound. Such voxels are considered sufficiently simple. Further decomposition is not only unnecessary but would be counter-productive, creating many smaller voxels to be stored and navigated. The *depth* criterion checks for a voxel at some maximum depth within the decomposition. Such voxels have not yet satisfied the simplicity criterion despite extensive decomposition. Further attempts at simplifying this voxel are given up as a bad job to avoid runaway.

Fig 6.3a: Ray Navigation of an Octtree: The SMART Algorithm

The Navigation Method

A ray is to be navigated in path length order through the leaf voxels in an octtree decomposition of a unit cube, from a known point in a known direction. This is achieved in a recursive sequence of vertical and horizontal steps within the octtree diagram. Each type of step is navigated according to the components of a vector known as a SMART or Spatial Measure for Accelerated Ray Tracing. Each vertical step locates the child octant of the current voxel containing the current ray point. The current voxel is initially the octtree root. Each subsequent horizontal step locates the next sibling navigated by the ray. The entry point into this sibling becomes the new ray point. The navigation of each sibling recurses down to the appropriate leaf descendants. The navigation is maintained incrementally with efficient recurrence relations using values already present in the current environment. It is optimised by quantisation to fixed point integer arithmetic.

Notation

The vertical and horizontal SMART navigators are denoted $\underline{V}$ and $\underline{H}$ respectively.

Consider the octant in which the ray's direction falls. In any internal voxel navigated by the ray, this octant contains the vertex intersection of the ray's exit planes from the voxel. This vertex is called the *exit vertex* and the octant is called the *exit octant*. Let the exit octant have absolute Morton digit 'exit' [Fig 5.6.2a]. The diagonally opposite octant contains the vertex intersection of the ray's entry planes into the voxel. This vertex is called the *entry vertex* and the octant is called the *entry octant*. Let the entry octant have absolute Morton digit 'entry' [Fig 5.6.2a].

Let the ray have a distance vector $\underline{\delta} = (x, y, z)$ of distances to traverse from its current point to the exit planes of the current voxel. All distances are strictly positive. This is simply the current ray point vector taken relative to the ray's exit vertex. Each horizontal step navigates the ray's current point to the first exit plane(s) struck of the current child. Let the ray have direction vector $\underline{\Delta} = (X, Y, Z)$ of relative traversal rates across each dimension. All rates are non-negative, and

the maximum is unitary.

The child of the current internal voxel containing the current ray point is indexed by its relative Morton digit with respect to the entry octant [Fig 5.6.2a]. Its absolute Morton digit is easily recovered from this [Fig 5.6.2a]. The bits of the relative digit are concatenated in the order ZYX. Let this relative Morton digit be 'relative_child'. This is initially found by the vertical SMART navigator $\underline{V}$ and is then updated by the horizontal SMART navigator $\underline{H}$.

Let each child of the current internal voxel have width 'w'. Let $S \in \{ 1, X, Y, Z \} \setminus \{ 0 \}$ be a context-dependent positive scaling factor. The three vectors $\underline{V} = S\underline{\delta}$, $\underline{H} = \underline{\Delta} \times \underline{\delta}$, $w\underline{\Delta}$ and the comparison variable Sw are forwarded to each stage of the navigation. New values after any step are denoted by a tilde superscript, e.g. $\tilde{\underline{V}}$ represents the updated vertical navigator $\underline{V}$. All values are maintained with an incremental recurrence relation.

Vertical Step Navigation

If the current voxel proves to be a leaf, the objects in its heterogeneous list are checked for the scene model's solution. Otherwise the current voxel is internal and the SMART navigation continues down to the appropriate leaves.

The relative Morton digit of the child containing the current ray point is found according to the vertical SMART navigator $\underline{V}$. Clearly

$$\text{relative\_child} = \left\{ \begin{array}{l} x \left\{ \begin{array}{l} \leq w \rightarrow 1 \\ > w \rightarrow 0 \end{array} \right\} \text{BIT\_OR} \\ y \left\{ \begin{array}{l} \leq w \rightarrow 2 \\ > w \rightarrow 0 \end{array} \right\} \text{BIT\_OR} \\ z \left\{ \begin{array}{l} \leq w \rightarrow 4 \\ > w \rightarrow 0 \end{array} \right\} \end{array} \right\}$$

Now $\underline{V} = S\underline{\delta}$ and since S is positive and scaling by a positive factor has no effect on ordering,

$$\text{relative\_child} = \left\{ \begin{array}{l} V_x = Sx \left\{ \begin{array}{l} \leq Sw \rightarrow 1 \\ > Sw \rightarrow 0 \end{array} \right\} \text{BIT\_OR} \\ V_y = Sy \left\{ \begin{array}{l} \leq Sw \rightarrow 2 \\ > Sw \rightarrow 0 \end{array} \right\} \text{BIT\_OR} \\ V_z = Sz \left\{ \begin{array}{l} \leq Sw \rightarrow 4 \\ > Sw \rightarrow 0 \end{array} \right\} \end{array} \right\}$$

Since $\underline{V}$ and Sw are forwarded directly by the recursive navigation, no multiplication is required for their calculation.

Navigation Maintenance down a Vertical Step

After each vertical step the new distance vector $\tilde{\underline{\delta}}$ is taken relative to the child's exit vertex. The new vectors $\tilde{\underline{V}}$, $\tilde{\underline{H}}$, $\tilde{w}\underline{\Delta}$ and comparison variable $S\tilde{w}$ are found by a recurrence from their previous values. The scaling factor S and direction vector $\underline{\Delta}$ remain unchanged.

The vertical step has been navigated by comparing each component of $\underline{V} = S\underline{\delta}$ with Sw. Consider the X dimension. If $Sx \leq Sw$ the child octant lies on the opposite side of the current voxel's X bisection plane to the entry octant. Then $\tilde{x} = x$ so $\tilde{V}_x = S\tilde{x} = Sx = V_x$. Similarly, there is no change in the contribution of $\tilde{x}$ to $\tilde{\underline{H}}$ compared with that of x to $\underline{H}$. If however $Sx > Sw$ the child octant lies on the same side of the X bisection plane as the entry octant. Then $\tilde{x} = x-w$ so

$$\tilde{V}_x = S\tilde{x} = S[x-w] = Sx-Sw = V_x-Sw$$

A difference arises between the contribution of $\tilde{x}$ to $\underline{H}$ compared to that of x to $\underline{H}$ such that

$$\tilde{\underline{H}} = \underline{\Delta} \times \tilde{\underline{\delta}} = (X, Y, Z) \times (x-w, y, z) = (Yz-Zy, Zx-Xz-Zw, Xy-Yx+Yw) = \underline{H} + (0, -w\Delta_z, w\Delta_y)$$

The same workings hold for any cyclic permutation of dimensions. The effect on each SMART navigator may be summed independently across dimensions to give the appropriate new values.

Since the width of each child octant is simply half that of the internal voxel, the new values of $\tilde{w}\underline{\Delta}$ and $S\tilde{w}$ are found by halving the previous values. This may be achieved with a bit shift in integer format, which is a particularly simple operation in binary computation.

Horizontal Step Navigation

Horizontal steps are navigated with the SMARTs maintained after the vertical step so that $\underline{V} := \tilde{\underline{V}}$ and $\underline{H} := \tilde{\underline{H}}$. The relative Morton digit of the next child navigated by the ray is found by setting the bits corresponding to the bisection planes traversed in the horizontal step navigated by the ray. These planes are found according to the horizontal navigator $\underline{H}$. The horizontal navigation is iterated between siblings and a recursive vertical step is navigated down each. This iteration continues until some of the bits to be set in the relative child digit are found to be already set. The corresponding bisection planes of the current internal node have then already been crossed, and so the recursion is complete within this voxel. This branch of the recursive navigation then terminates, causing an automatic ascending vertical step to the parent internal node.

The bisection planes traversed by each horizontal step are the first planes simultaneously struck by the ray is it leaves the current child. These are found by considering the order in which the three bisection planes are struck.

Consider the order in which the Y and Z exit planes are struck (if at all) as the ray leaves the current child. Let 'Y before Z', 'Y with Z' and 'Y after Z' denote the Y plane being struck before the Z plane, simultaneously with the Z plane, and after the Z plane respectively. Suppose that the Y plane *is* struck. Let $r_z = z - y\frac{Z}{Y}$ be the distance remaining to traverse across the Z dimension to the Z plane at the intersection with the Y plane. Clearly, the Y plane is struck *before* the Z plane when this distance is positive, so that

$$z - y\frac{Z}{Y} = r_z > 0 \leftrightarrow \text{Y before Z}$$

Now Y is known to be positive, being non-negative and moreover non-zero since the Y plane is struck. Since scaling by a positive factor has no effect on sign,

$$Yz - yZ = Yr_z > 0 \leftrightarrow \text{Y before Z}$$

Suppose instead that the Z plane *is* struck. Then a similar equivalence may be derived

$$Zy - zY = Zr_y > 0 \leftrightarrow \text{Y after Z}$$

Now

$$\underline{H} = \underline{\Delta} \times \underline{\delta} = (X, Y, Z) \times (x, y, z) = (Yz - yZ, Zx - zX, Xy - xY)$$

Fig 6.3a

and so $H_x = Yz - yZ \begin{cases} = Yr_z \\ = -Zr_y \end{cases}$. Therefore

$$H_x \begin{cases} > 0 \rightarrow \begin{cases} Yr_z > 0 \\ Zr_y < 0 \end{cases} \rightarrow \text{Y before Z} \\ = 0 \rightarrow \begin{cases} Yr_z = 0 \\ Zr_y = 0 \end{cases} \rightarrow \text{Y with Z} \\ < 0 \rightarrow \begin{cases} Yr_z < 0 \\ Zr_y > 0 \end{cases} \rightarrow \text{Y after Z} \end{cases}$$

If the ray is parallel to the X axis, neither Y nor Z plane is struck. In this case $H_x = 0$ indicating that they are struck simultaneously, at infinity.

The same workings hold for any cyclic permutation of dimensions. The sign of each component in $\underline{H}$ therefore indicates which of the planes in the other two dimensions is struck first. The plane(s) traversed by the ray on leaving the current child are found by examining appropriate components of $\underline{H}$. Superficially there appear to be $3^3 = 27$ possible component sign states. However, many of these infer cyclic inconsistencies such as 'X before Y' yet 'Y before Z' and 'Z before Y'. Such inconsistent states never actually arise. There are only thirteen consistent sign configurations, each *balanced* in the sense that *either* all components are zero *or* both a positive and negative component exist.

The relative Morton digit of the next sibling navigated is determined through a decision tree requiring only two component sign examinations:

$$H_x \begin{cases} > 0 \rightarrow \text{Y before Z ;} H_z \begin{cases} > 0 \rightarrow \text{X before Y: X first; relative\_child BIT\_OR-ED } 001_2 \\ = 0 \rightarrow \text{X with Y: X, Y first; relative\_child BIT\_OR-ED } 011_2 \\ < 0 \rightarrow \text{X after Y: Y first; relative\_child BIT\_OR-ED } 010_2 \end{cases} \\ = 0 \rightarrow \text{Y with Z ;} H_z \begin{cases} > 0 \rightarrow \text{X before Y: X first; relative\_child BIT\_OR-ED } 001_2 \\ = 0 \rightarrow \text{X with Y: X, Y, Z first; relative\_child BIT\_OR-ED } 111_2 \\ < 0 \rightarrow \text{X after Y: Y, Z first; relative\_child BIT\_OR-ED } 110_2 \end{cases} \\ < 0 \rightarrow \text{Y after Z ;} H_y \begin{cases} > 0 \rightarrow \text{Z before X: Z first; relative\_child BIT\_OR-ED } 100_2 \\ = 0 \rightarrow \text{Z with X: X, Z first; relative\_child BIT\_OR-ED } 101_2 \\ < 0 \rightarrow \text{Z after X: X first; relative\_child BIT\_OR-ED } 001_2 \end{cases} \end{cases}$$

The relative Morton digit is updated by setting the bits corresponding to the planes traversed in the horizontal step. If any of these bits are already set, the navigation has exhausted the current

internal voxel and this branch of the recursion is terminated. Otherwise the current ray point is updated to the intersection with the traversed planes within this next sibling.

Navigation Maintenance across Horizontal Steps

After a horizontal step the new distance vector $\tilde{\underline{\delta}}$ is the updated current ray point relative to the new sibling's exit vertex. The new vectors $\tilde{\underline{V}}$, $\tilde{\underline{H}}$ and the comparison variable $\tilde{S}w$ are found by recurrence relations from their previous values. The sibling width w and direction vector $\underline{\Delta}$ remain unchanged, as does the vector $w\underline{\Delta}$. The scaling factor $\tilde{S}$ is updated after a horizontal step to correspond to one of the bisection planes traversed.

There are three cases of maintenance to consider after a horizontal step. Each corresponds to a different number of planes traversed.

- Suppose that exactly one plane is traversed in the horizontal step.

  Let this be the X plane. Clearly

$$\tilde{\underline{\delta}} = \left(w,\ y - x\frac{Y}{X},\ z - x\frac{Z}{X} \right)$$

  By selecting $\tilde{S} = X$ all undesirable division is removed from the new vertical SMART navigator, so that

$$\tilde{\underline{V}} = \tilde{S}\tilde{\underline{\delta}} = (wX,\ Xy - xY,\ Xz - xZ) = (w\Delta_x,\ H_z,\ -H_y)$$

  This is to be compared with the new comparison variable

$$\tilde{S}w = Xw = w\Delta_x$$

  The new horizontal SMART navigator is given by

$$\tilde{\underline{H}} = \underline{\Delta} \times \tilde{\underline{\delta}} = (X, Y, Z) \times \left(w,\ y - x.\frac{Y}{X},\ z - x.\frac{Z}{X} \right)$$

$$= \left(\left[Yz - Yx.\frac{Z}{X} \right] - [Zy - Zx.\frac{Y}{X}],\ Zw - [Xz - xZ],\ [Xy - xY] - Yw \right)$$

$$= (Yz - Zy,\ Zx - zX + Zw,\ Xy - xY - Yw)$$

$$= \underline{H} + (0,\ w\Delta_z,\ -w\Delta_y)$$

  Conveniently, the X divided terms in the first component cancel to avoid undesirable division once more.

This maintenance employs values already in the current environment. Negation is the only arithmetic required. The same workings hold for any cyclic permutation of dimensions.

- Suppose instead that exactly two planes are traversed in the horizontal step.

  Let these be the Y and Z planes. The ray then exits through an edge of the child. Clearly,

$$\underline{\tilde{\delta}} \begin{cases} = (x - z\frac{X}{Z}, w, w) \\ = (x - y\frac{X}{Y}, w, w) \end{cases}$$

  By selecting $\tilde{S} = Z$ all undesirable division is removed from the updated vertical SMART navigator, so that

$$\underline{\tilde{V}} = \tilde{S}\underline{\tilde{\delta}} = (Zx - zX, wZ, wZ) = (H_y, w\Delta_z, w\Delta_z)$$

  This is to be compared with the new comparison variable

$$\tilde{S}w = Zw = w\Delta_z$$

  The new horizontal SMART navigator is given by

$$\underline{\tilde{H}} = \underline{\Delta} \times \underline{\tilde{\delta}} \begin{cases} = (X, Y, Z) \times (x - y.\frac{X}{Y}, w, w) \\ = (X, Y, Z) \times (x - z.\frac{X}{Z}, w, w) \end{cases}$$

$$\begin{cases} = (Yw - Zw, [Zx - Zy.\frac{X}{Y}] - Xw, Xw - [Yx - yX]) \\ = (Yw - Zw, [Zx - zX] - Xw, Xw - [Yx - Yz.\frac{X}{Z}]) \end{cases}$$

$$= (Yw - Zw, [Zx - zX] - Xw, [Xy - xY] + Xw)$$

$$= (w\Delta_y - w\Delta_z, H_y - w\Delta_x, H_z + w\Delta_x)$$

  The same workings hold for any cyclic permutation of dimensions.

- Suppose finally that exactly all three planes are traversed in the horizontal step.

  The ray then exits through the child node's exit vertex. Clearly

$$\underline{\tilde{\delta}} = (w, w, w)$$

  Arbitrarily selecting $\tilde{S} = X$, the updated vertical SMART navigator becomes

$$\underline{\tilde{V}} = \tilde{S}\underline{\tilde{\delta}} = (Xw, Xw, Xw) = (w\Delta_x, w\Delta_x, w\Delta_x)$$

  This is to be compared with the new comparison variable

$$\tilde{S}w = Xw = w\Delta_x$$

Fig 6.3a

The new horizontal SMART navigator is given by

$$\underline{\tilde{H}} = \underline{\Delta}\times\underline{\tilde{\delta}} = (X, Y, Z)\times(w, w, w)$$

$$= (Yw-Zw, Zw-Xw, Xw-Yw)$$

$$= (w\Delta_y-w\Delta_z, w\Delta_z-w\Delta_x, w\Delta_z-w\Delta_y)$$

So no matter whether one, two or all three planes are traversed in the horizontal step, the SMART navigation is maintained across a horizontal step by the simple addition or subtraction of values already in the current environment. Moreover, if the navigation is initially quantised over the integers then no subsequent rounding errors can be incurred since there is no division. The navigation may therefore be maintained in efficient integer arithmetic.

Fixed Point Quantisation to Optimise Navigation

All steps are navigated according to $\underline{V} = \underline{\Delta}\times\underline{\delta}$, $\underline{H} = S\underline{\delta}$, $w\underline{\Delta}$ and Sw where $S \in \{ 1, X, Y, Z \} \setminus \{ 0 \}$. The navigation may be maintained with efficient fixed point arithmetic if these always fit in the appropriate fixed point format. Consider a 32 bit signed format as an example, storing integers from the range $[-2^{31},2^{31})$. All vectors will fit into this format provided their infinity norms lie below a critical bound, 2^{31} in this example.

The direction vector $\underline{\Delta}$ is fixed, whilst $\underline{\delta}$ and w assume their maximum values at the root level. Since $S \in \{ 1, X, Y, Z \} \setminus \{ 0 \}$ then $|S| \le |\underline{\Delta}|_\infty$. For any child within an octtree of root voxel width R, both $|\underline{\delta}|_\infty \le R$ and $|w| \le R$. So for any octtree voxel,

$$|\underline{V}|_\infty = |S\underline{\delta}|_\infty = |S|\,|\underline{\delta}|_\infty \le |\underline{\Delta}|_\infty R$$

$$|\underline{H}|_\infty = |\underline{\Delta}\times\underline{\delta}|_\infty \le |\underline{\Delta}|_\infty|\underline{\delta}|_\infty \le |\underline{\Delta}|_\infty R$$

$$|w\underline{\Delta}|_\infty = |w|\,|\underline{\Delta}|_\infty \le |\underline{\Delta}|_\infty R$$

$$|Sw| = |S|\,|w| \le |\underline{\Delta}|_\infty R$$

Therefore all of $|\underline{V}|_\infty$, $|\underline{H}|_\infty$, $|w\underline{\Delta}|_\infty$ and $|Sw|$ are bounded above by $|\underline{\Delta}|_\infty R$. This bound is uniform across all rays. All values will fit as required if both $|\underline{\Delta}|_\infty$ and $R < 2^{15}$. This is easily ensured when representing real numbers in a mantissa/exponent floating point format such as the IEEE standard. The root voxel and ray direction are scaled with a single floating point

initialisation per ray to make both the root voxel's width and the ray direction's infinity norm become rational unity. The fourteen most significant bits are extracted from the relevant variable's mantissa with efficient bit masks and shifts. This provides a resolution of $2^{14} = 16384$ graduations. The navigation may proceed after this quantisation under incremental fixed point arithmetic.

Invoking the SMART Navigation

The SMART navigation is invoked by an initial vertical step down the root voxel with scaling factor S = 1. The vertical SMART $\underline{V}$ is then the unscaled distance vector $\underline{\delta}$, and the comparison variable is the unscaled child width w. This is the canonical choice and avoids the introduction of any degenerate zero scaling factor. The unit scaling factor is used until a horizontal step is navigated. Horizontal steps identify new siblings for vertical navigation. The scaling factor used in any such vertical navigation depends on the bisection planes traversed horizontally in a manner avoiding degenerate zero scaling factors once more.

Fig 6.3b: A 'C' Code Implementation of the SMART Algorithm

```
/* ------------------------------------------ DEFINITIONS ------------------------------------------ */

extern int oct_tree_node[];                                     /* Autumnal Octtree list */
extern unsigned char entry_octant;                              /* Morton digit of entry octant */

struct discrete_vector { int x, y, z ; } ;

#define BISECT(bit,a,b,c)                                                                   \
    if (V_smart.a <= V_compare)                                                             \
    {   V_smart.a -= V_compare; H_smart.b -= wid_dir.c; H_smart.c += wid_dir.b; }           \
    else                                                                                    \
        relative_child |= bit

#define ONE_PLANE_STRUCK(bit,a,b,c)                                                         \
    if (relative_child & bit) break; else relative_child |= bit;                            \
    V_compare = child_wid_dir.a;                                                            \
    V_smart.a = wid_dir.a; V_smart.b = H_smart.c; V_smart.c = -H_smart.b;                   \
    H_smart.b += wid_dir.c; H_smart.c -= wid_dir.b;

#define X_PLANE_STRUCK ONE_PLANE_STRUCK(1,x,y,z)
#define Y_PLANE_STRUCK ONE_PLANE_STRUCK(2,y,z,x)
#define Z_PLANE_STRUCK ONE_PLANE_STRUCK(4,z,x,y)

#define TWO_PLANE_STRUCK(bits,a,b,c)                                                        \
    if (relative_child & bits) break; else relative_child |= bits;                          \
    V_compare = child_wid_dir.c;                                                            \
    V_smart.a = H_smart.b; V_smart.b = wid_dir.c;  V_smart.c = wid_dir.c;                   \
    H_smart.a = wid_dir.b - wid_dir.c; H_smart.b -= wid_dir.a; H_smart.c += wid_dir.a;

#define Y_Z_PLANE_STRUCK TWO_PLANE_STRUCK(6,x,y,z)
#define Z_X_PLANE_STRUCK TWO_PLANE_STRUCK(5,y,z,x)
#define X_Y_PLANE_STRUCK TWO_PLANE_STRUCK(3,z,x,y)

#define THREE_PLANE_STRUCK                                                                  \
    if (relative_child & 7) break; else relative_child |= 7;                                \
    V_compare = child_wid_dir.x;                                                            \
    V_smart.x = wid_dir.x; V_smart.y = wid_dir.x; V_smart.z = wid_dir.x;                    \
    H_smart.x = wid_dir.y - wid_dir.z; H_smart.y = wid_dir.z - wid_dir.x; H_smart.z = wid_dir.x - wid_dir.y;

#define X_Y_Z_PLANE_STRUCK THREE_PLANE_STRUCK

/* ------------------------------------- THE PROGRAM PROPER ------------------------------------- */

void traverse( index, V_smart, V_compare, H_smart, wid_dir )    /* Navigates a ray through an oct-tree */
int index;                                                      /* Parent index */
struct discrete_vector V_smart;                                 /* Vertical navigator */
unsigned int V_compare;                                         /* Vertical navigator comparison */
struct discrete_vector H_smart;                                 /* Horizontal navigator */
struct discrete_vector wid_dir;                                 /* Ray direction scaled by child width */
{   struct discrete_vector child_wid_dir;                       /* Vertically propagated */
    unsigned char relative_child = 0;                           /* Relative Morton digit */

    if (LEAF(index))  { /* Node is a leaf .. query heterogeneous object list */ }
    else                                                        /* Node is internal */
    {   BISECT(1,x,y,z); BISECT(2,y,z,x); BISECT(4,z,x,y);      /* First perform vertical step */

        V_compare >>= 1;                                        /* Half values for vertical propagation */
        child_wid_dir.x = wid_dir.x>>1; child_wid_dir.y = wid_dir.y>>1; child_wid_dir.z = wid_dir.z>>1;

        traverse
        (   oct_tree_node[index+(relative_child^entry_octant)],
            V_smart, V_compare, H_smart, child_wid_dir
        );

        while (relative_child !=7 )                             /*Then horizontal steps */
        {   if (H_smart.x < 0)
            {   if (H_smart.y < 0) { X_PLANE_STRUCK } else
                if (H_smart.y > 0) { Z_PLANE_STRUCK } else
                /* (H_smart.y == 0) */   { Z_X_PLANE_STRUCK }
            }
            else
            if (H_smart.x > 0)
            {   if (H_smart.z < 0) { Y_PLANE_STRUCK } else
                if (H_smart.z > 0) { X_PLANE_STRUCK } else
                /* (H_smart.z == 0) */   { X_Y_PLANE_STRUCK }

            }
            else
            /* (H_smart.x == 0) */
            {   if (H_smart.y < 0) { X_PLANE_STRUCK } else
                if (H_smart.y > 0) { Y_Z_PLANE_STRUCK } else
                /* (H_smart.y == 0) */   { X_Y_Z_PLANE_STRUCK }
            }
            traverse
            (   oct_tree_node[index+(relative_child^entry_octant)],
                V_smart, V_compare, H_smart, child_wid_dir
            );
        }
    }
}
```

These twin termination criteria may be predetermined by subjective choice or refined during image synthesis. They will effect both the efficiency of a ray's solution to the scene model and the octtree's storage requirements.

6.4.2. Assigning the Heterogeneous Object List to a Voxel

A heterogeneous object list is assigned to each leaf voxel in an octtree with the recursive method described for each cell in a grid partition [Section 5.4.4]. Heterogeneous objects from the parent's list are once more checked for being heterogeneous to each child with interval analysis [Section 5.8]. Octtrees may be dynamically generated by lazy construction during image synthesis.

6.5. Data Structures for the Representation of an Octtree

The octtree is stored in Autumnal format [Fabbrinni,Montani;1986] as for ARTS. Internal nodes are represented by an array of eight records, one per child in Morton order. Each record comprises a pair of fields. These are a status field and pointer field. The status defines the list indexed by the pointer. Only two cases of status are distinguished for the SMART implementation, as opposed to four by ARTS. These are LEAF status for leaf children and INTERNAL for internal children. Each pair of fields is encoded in a single integer. The status field occupies the top bit whilst the pointer occupies the remainder. Internal children reference back into the Autumnal tree list whilst leaf children reference heterogeneous object lists within a global array by the double indirection previously described for grid partition cells [Section 5.5]. Leaf children in either homogeneous state reference an empty list in this array. Their precise homogeneous state is immaterial to a ray's scene model solution and is not stored.

6.6. Storage Considerations

Heterogeneous list duplications between leaves are all made to reference the same entry within the global array rather than separate copies. This minimises the global array's size and is accomplished as before [Section 5.6].

Intuitively the octtree seems a more compact decomposition than the grid partition. However the ARTS paper suggested the grid partition's relative inefficiency in space requirements was insignificant when compared to the greater efficiency of navigation schemes proposed therein. It was claimed that "*whilst octtree representations can be expected to take advantage of the spatial coherence found in most objects ... even if this is generally the case it will be difficult to predict the resolution at which the octtree encoded structure becomes superior*". They asserted that "*this observation is most important because in general it is more time consuming to retrieve information from and traverse the octtree*" and that "*even at high resolutions it is possible to envisage a scene with many objects and low homogeneity for which the octtree structure will not necessarily be justified*". Reference was made to the quadtree complexity theorem [Hunter,Steiglitz;1980] but its consequences were not fully elaborated.

The quadtree complexity theorem may be generalised for 3D octtree decompositions with a simplicity criterion for termination of zero objects. A voxel is then decomposed if believed to contain even a single object surface. The theory essentially provides a bounding function for the total octtree node count in terms of the maximum decomposition depth and the scene model. This function is a constant multiple of the sum of this depth and the number of voxels at this depth containing an object surface. The constant is in the order of tens. For all but the most pathologically sparse scenes, any non-trivial decomposition depth is dwarfed by this voxel count and so the latter summand is dominant. An octtree decomposition of a sparse scene is clearly more compact than a grid partition anyway. The surfaces of smooth objects without fractal effects appear locally planar within any voxel whose depth is sufficient to make its size insignificant compared to these objects. An extra level of decomposition therefore tends to quadruple this leaf voxel count and hence the total node count by the complexity theory. The size of an octtree generated with a less demanding simplicity termination criterion will be smaller still and will eventually become constant if no objects overlap.

This compares favourably to the size of grid partition which increases by a constant factor of eight for every extra level of decomposition. The size of each scene partition rises exponentially with decomposition depth, but the grid partition's size multiplies at a rate of $8 = 2^3$ whilst the octtree tends to multiply at a rate of $4 = 2^{3-1}$. This one dimensional decrease in the complexity of the octtree compared to the grid partition is the fundamental result of the octtree complexity theory.

Even in the worst case the storage requirement of an octtree decomposition is less than a seventh or 14% more than that for a grid partition of the same scene [Fig 6.6a]. Such scenes exhibit no spatial coherency at any level above the maximum decomposition depth since the simplicity termination criterion is never satisfied before reaching this depth. It is a strange scene indeed in which *every* voxel one level above a non-trivial maximum depth contains a single object surface, let alone several. Any such scene would be full of object *surfaces*. If the maximum depth were sufficient to make the size of leaf voxels insignificant compared to that of most objects, these objects would form a nearly contiguous mass which is hardly a usual scene model. A typical scene will contain extensive spatially coherent regions for any non-trivial decomposition depth.

6.6.1. The Lazy Construction of Octtrees

The above comparison in storage requirements only considers decompositions which are fully constructed before ray tracing. As previously remarked however, dynamic generation by lazy construction is an attractive alternative [Section 5.4.3]. Octtrees are particularly amenable to lazy construction. As for grid partitions, lazy construction offers opportunities for computational savings. Unlike grid partitions however, the lazy construction of octtrees also offers opportunities for storage savings. Whereas the cells of a grid partition are listed in given raster order, the internal nodes of an octtree may be listed in any convenient order [Section 6.2.2]. Octtree storage need only be allocated as each internal node is generated, maintaining a contiguous block of used memory without wasted holes. Since a grid partition is held in given raster order however, the entire storage between any two navigated cells must be allocated including those which have not yet been navigated. The

Fig 6.6a: Relative Storage Requirements of the Octtree and Grid Decomposition

The octtree decomposition ignores homogeneous regions of a scene to focus onto object surfaces. It therefore adapts to local scene complexity and may often require less storage than an indiscriminate grid partition over all scene regions.

An equation is derived to indicate when such situations arise. This also shows that even in the worst case of a totally degenerate scene, the octtree requires just under one seventh more storage than the grid partition.

Consider the decomposition of a given scene's bounding voxel to maximum depth 'D'. The storage requirement for an octtree will depend on the distribution of leaf voxels through the levels down to this depth.

Let $\{L_i\}_{i=0}^{D}$ be the set of leaf voxel counts at each level. The root is at level 'D', the root's children at 'D-1', the grandchildren at 'D-2', and so on down to level '0' of maximum resolution.

The voxel leaves partition the grid of 8^D cells over the scene's bounding voxel. The leaf counts are therefore constrained by the relation $8^D = \sum_{i=0}^{D} 8^i L_i$. Let the total leaf count be $L = \sum_{i=0}^{D} L_i$ and the total internal node count be I.

Let Octtree(D) and Grid(D) be the storage requirements of an octtree and grid partition respectively. Storage is measured by total pointer count.

In Autumnal format, Octtree(D) = 8I. Consider constructing the octtree bottom-up from a pool of nodes initialised with the leaves. A group of eight siblings is removed and merged to an internal node which is returned. This decrements the pool count by seven, and is repeated until only the octtree root remains. Then $L-7I = 1 \rightarrow I = \frac{L-1}{7}$. Therefore

$$\text{Octtree(D)} = \frac{8(L-1)}{7} = \frac{8}{7}\left(\sum_{i=0}^{D} L_i - 1\right)$$

which varies with the leaf distribution.

Clearly however, Grid(D) = 8^D is constant for a given maximum decomposition depth.

These requirements may be compared by finding the extra storage of the octtree as a fraction of the grid's storage. Then

$$\frac{\text{Octtree(D)} - \text{Grid(D)}}{\text{Grid(D)}} = \frac{8(\sum_{i=0}^{D} L_i - 1) - 7.8^D}{7.8^D} = \frac{\sum_{i=0}^{D} L_i - (1+\frac{7}{8}8^D)}{\frac{7}{8}8^D}$$

The octtree is the more compact decomposition if this is negative. The leaf count distribution will then be such that

$$\sum_{i=0}^{D} L_i < 1+\frac{7}{8}8^D$$

Consider the worst case for an octtree of maximal extra storage requirement relative to a grid partition. This clearly arises when leaf voxels are only present at level zero. Then

$$L_0 = 8^D;\quad L_1,L_2,\ldots,L_D = 0 \rightarrow \frac{\text{Octtree(D)} - \text{Grid(D)}}{\text{Grid(D)}} = \frac{\frac{1}{8}8^D - 1}{\frac{7}{8}8^D} = \frac{1}{7} - \frac{1}{\frac{7}{8}8^D} < \frac{1}{7}$$

The octtree's storage requirement is then just under a seventh greater that of a grid. Such a scene would have extremely low spatial coherency.

Consider instead the general case. From the leaf distribution constraint,

$$8^D = \sum_{i=0}^{D} L_i 8^i = L_0 + \sum_{i=1}^{D} L_i 8^i \geq L_0 + 8\sum_{i=1}^{D} L_i$$

$$\rightarrow \frac{7L_0+8^D}{8} \geq \sum_{i=0}^{D} L_i$$

So from the above condition, the octtree will be the more compact decomposition if

$$\frac{7L_0+8^D}{8} < 1+\frac{7}{8}8^D$$

The octtree is therefore the more compact decomposition if the leaf count at level zero is below a guaranteed break even point such that

$$L_0 < \frac{8+6.8^D}{7} = \text{guaranteed\_break\_even(D)}$$

This guaranteed break even point increases at a constant exponential rate of eight with maximum decomposition depth.

All voxels at level zero are leaves. None of their predecessors are believed to contain fewer object surfaces than demanded by the simplicity criterion. Consider how this leaf count behaves with increasing maximum decomposition depth. At first, all eight children spawned from a voxel may still fail the simplicity criterion. The count may then well grow at an exponential rate of eight in parallel with the guaranteed break even point. However, a stage will eventually be reached where voxels of high resolutions are significantly smaller than the scene objects. Provided the object surfaces are smooth and without fractal characteristics they will appear locally planar within these voxels. Several of the children spawned from such a voxel will finally satisfy the simplicity criterion. The growth of the leaf count at level zero will then tail off, and may indeed stop for an undemanding simplicity criterion. At some critical depth this leaf count will fall below the guaranteed break even point, since the latter keeps growing at an exponential rate of eight. The octtree will therefore always be a more compact decomposition beyond some critical maximum depth for a scene containing such objects.

Fig 6.6a

storage array for a grid partition would usually be allocated before tracing the first ray. Extensive sections of this array corresponding to cells never actually navigated would remain unused and constitute a waste of storage. The dynamic generation of a grid partition would have to hold the octtree decomposition throughout ray tracing to dynamically assign voxel lists. Since the octtree may be discarded after preprocessing a grid partition, the lazy construction of a grid partition would actually require more storage than preprocessing.

The octtree's greater flexibility offers not only the opportunity of dynamic generation but also dynamic refinement. The computational costs of ray tracing leaf voxels may prove unacceptably high if these regions have not been made sufficiently simple. If so, the termination criteria can be made more demanding during ray tracing to increase local scene simplification and decrease future costs. This is easy for the arbitrarily ordered octtree which may be supplemented with the refined internal nodes as generated. Some researchers commence ray tracing with a single leaf voxel containing the entire scene which is progressively decomposed during ray tracing [Lathrop;1988]. Dynamic refinement is difficult for the grid partition however since this must be maintained in given raster order. An entire cell array would have to be reallocated for any refined resolution.

6.6.2. Efficient Use of Memory

Technological developments are providing an on-going increase in available memory. This extra storage allows scene models to be decomposed to ever more extensive depths. The octtree decomposition makes better use of storage than the grid partition at these greater depths. The octtree is efficiently navigated by the new SMART algorithm and therefore seems the more attractive decomposition in such circumstances.

Chapter 7: The Success of Implementations

Synopsis:

Chapter seven describes the success of the proposed acceleration techniques when implemented. Experimentally measured data are presented for the synthesis of four case studies, together with predictions of performance for more general scenes. The question of which acceleration technique is preferable is addressed in the light of these findings.

7.1. A Test Bed for Acceleration Techniques

A classical naive ray tracer [Whitted;1980] was implemented from scratch as a test-bed for the proposed acceleration techniques. It was written in the programming language 'C', a considerable undertaking which resulted in over 170k of source code. The host machine was a High Level Hardware Orion mini-computer running under the UNIX 4.2 BSD operating system. This machine was upgraded from an Orion-1 to an Orion-1/05 with a CLIPPER 32-bit microprocessor during this research. As a rough guide, the former is comparable in power to a VAX 11/750 and the latter to a VAX 8600. All experimental results given in this chapter are for the Orion-1/05. The source code should easily transfer to other machines since it does not rely on any graphics kernal such as GKS. Textual scene and view model specifications [Fig 2.1a;2.1b] are accepted as input to synthesise a raster image in a simple 24-bit pixel array with a standard header as output.

7.2. The Implementation of Acceleration Techniques

Three upgraded versions of this ray tracer have been fully implemented to test the proposed acceleration techniques. These exploit the Huffman derived bounding volume hierarchy [Chapter 4], grid partition [Chapter 5] and octtree decomposition [Chapter 6] in turn. This software has also been used extensively by other researchers at Bath with a requirement for realistic image synthesis. For example, 420 frames have been synthesised for an animated film of several hundred objects [John;1989], as have many images of a digitised face model containing several hundred polygons [Patel;1989].

These implementations work in two stages. The first stage converts the scene and view models from textual to a machine format and automatically generates an appropriate scene decomposition. The second stage synthesises an image by navigating this decomposition for accelerated ray tracing. Lazy construction has not yet been implemented due to limitations in time, but where applicable this should not present any problems.

This division of tasks separates the costs of constructing a decomposition from those of navigation, allowing an independent evaluation of each. It also proves convenient for the

synthesis of several views of a static scene. For example, in the synthesis of several frames for an animated 'fly-by' of a static scene only the view model need be reprocessed, avoiding the costs of repeated scene decomposition.

7.3. Criteria for Assessing the Success of the Proposed Acceleration Techniques

Any acceleration technique is successful if the savings gained exceed any extra overheads incurred. Scene decompositions provide savings in synthesis times by reducing the average number of objects queried by each ray and hence scene model solution time per ray. All three proposed decompositions proved to reduce synthesis times by orders of magnitude. Overheads are incurred both in computation for the construction and navigation of a decomposition and in memory for storage. The relative importance of these costs will depend on the local environment.

Differences in synthesis times between an accelerated technique and naive ray tracing can arise from many factors outside the scope of a scene decomposition. The scene model is locally simplified by a decomposition to reduce the number of objects queried by each ray. However, the degree of simplification also depends on the number of objects in the entire scene and their size. Moreover, scene model solution times per ray depend not only on the number of object queries, but also the objects' geometric complexity, the cost of navigating any scene decomposition and of course the rate of computation on the host machine. Total synthesis times also depend heavily on the number of rays traced which is outside the scope of these scene decompositions. Primary view ray count increases with an image's pixel count and the number of samples taken per pixel. The number of view rays spawned over successive generations increases with primary view ray count, the coverage of reflective and refractive objects in the scene, and the shading tree's cut-off depth. Illumination ray count varies with view ray count, the coverage of object surfaces, the number of light sources and the degree of culling by fine tuning techniques.

Before assessing the degree of acceleration obtained by navigating a scene decomposition over naive ray tracing by difference in synthesis times, any disparity must be known to arise from the use of the scene decomposition rather than any of these other factors. This

may be ensured by comparing synthesis times for the *same* images on the *same* hardware, so that all other factors remain constant. The degree of acceleration gained with a given scene decomposition may then be measured by expressing the synthesis time with the decomposition as a percentage of that for naive ray tracing. This is called the image's *relative synthesis time* under the decomposition.

7.4. Experimental Measurements for Four Case Studies

Experimental measurements are presented for four case studies [Fig 7.4a-d]. In each case the test image is synthesised at 512×512 pixels by navigating the three proposed scene decompositions. The acceleration of image synthesis proved so great that control times for naive ray tracing were impractical to measure at this resolution. Whilst accelerated synthesis never took more than a few hours, attempts at naive ray tracing often remained incomplete even after running for several days as a background job. Other research applications on the Orion often caused a crash or necessitated a reboot before this naive synthesis could finish. However, average synthesis time per traced ray may be expected to be independent of image size. Reliable estimates of naive ray tracing times at 512×512 pixel resolution are therefore presented as times measured for smaller 32×32 pixel images and scaled in proportion to image size. Experimental verification for accelerated image synthesis indicates that such estimates are accurate to within a few percent.

The four scenes are of varying object count and hence scope for local scene simplification. The most complex scene contains 7832 objects [Fig 7.4a] and is chosen from a proposed data base of test pieces [Haines;1987]. The next two scenes contain 5555 objects [Fig 7.4b] and 320 objects [Fig 7.4c] respectively, and were constructed to test the ray tracer implementations. The least complex scene contains 106 objects [Fig 7.4d] and is from a real application [John;1989].

Whilst only one Huffman-derived bounding volume hierarchy may be automatically generated from a given scene, many different grid partitions and octtree decompositions may be generated offering different degrees of local scene simplification. There is a 1-D spectrum of possible grid partitions, parameterised by cell depth in the decomposition, and

Fig 7.4a: Case Study 1
Experimental Results for Image Synthesis

Case Study 1	7832 Objects		1012542 Rays Traced	
Acceleration Technique	Cost			
	Time (days:hours:minutes:seconds) *(% Column Maximum)*			Storage (Kbytes) *(% Col. Max.)*
	Construction	Image Synthesis	Construction + Synthesis	Disk Space
Naive *(Estimate)*	*n/a*	66:00:00:00 *100*	66:00:00:00 *100*	*n/a*
Bounding Volume Hierarchy	1:39:14 *100*	2:34:09 *0.16*	4:13:23 *0.27*	354 *14*
Grid Partition *Depth 6*	4:02 *4.1*	3:16:16 *0.21*	3:20:18 *0.21*	1143 *44*
Octtree *Simplicity 1*† *Depth 9*	20:45 *21*	1:50:19 *0.12*	2:11:04 *0.14*	2596 *100*

† A voxel is decomposed only if its heterogenous list length exceeds 1

Case Study 1: The 7382 Object Scene

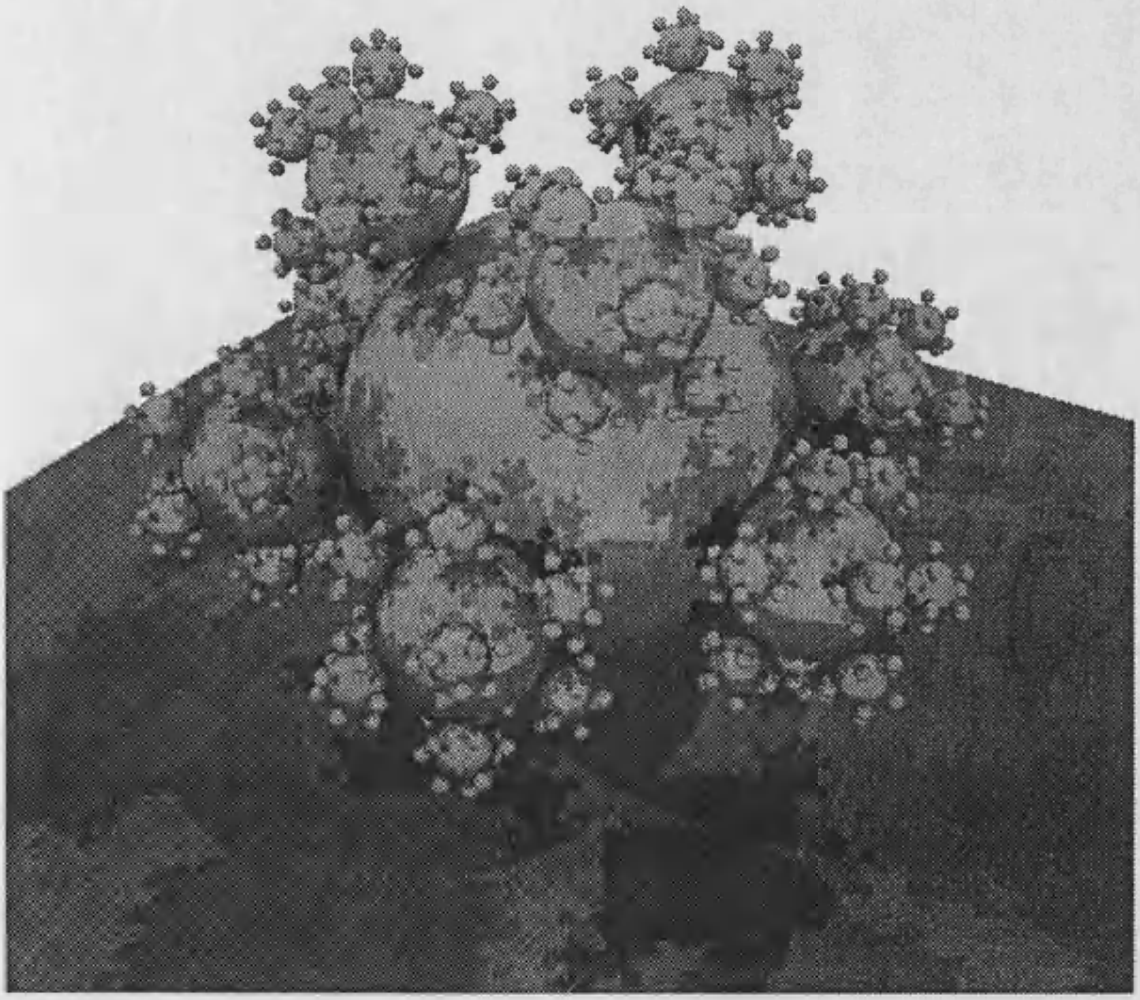


Fig 7.4a

Case Study 1: A Perspective View of the Bounding Volume Hierarchy

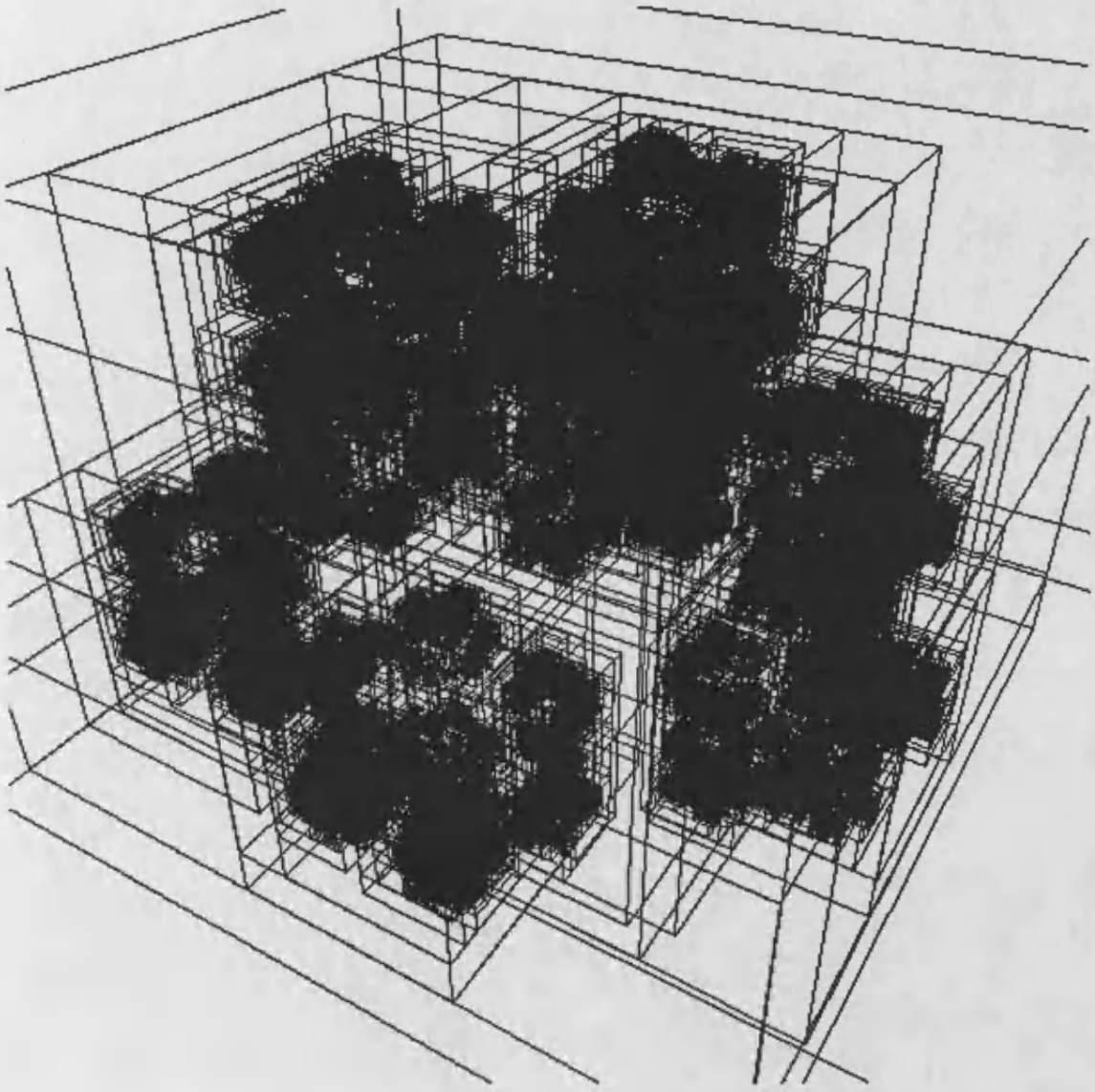


Fig 7.4a

Case Study 1: A Parallel Projection of Non Empty Cells in the Grid Partition

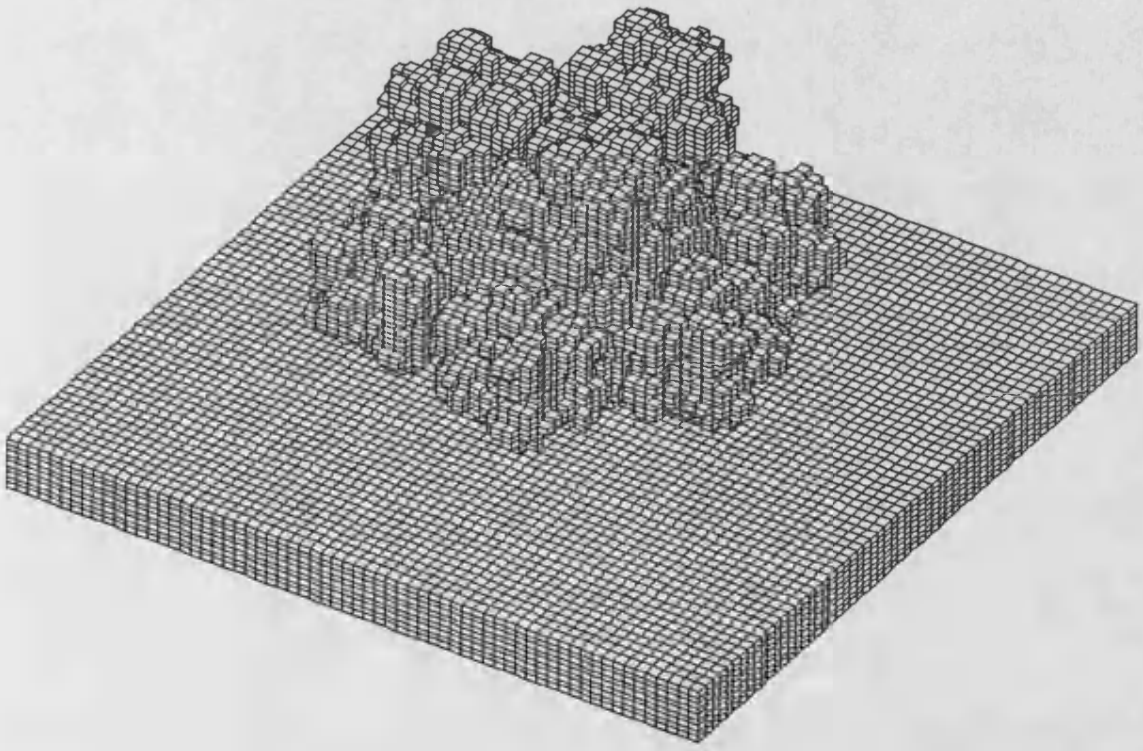


Fig 7.4a

Case Study 1: A Parallel Projection of Non Empty Voxels in the Octtree Decomposition

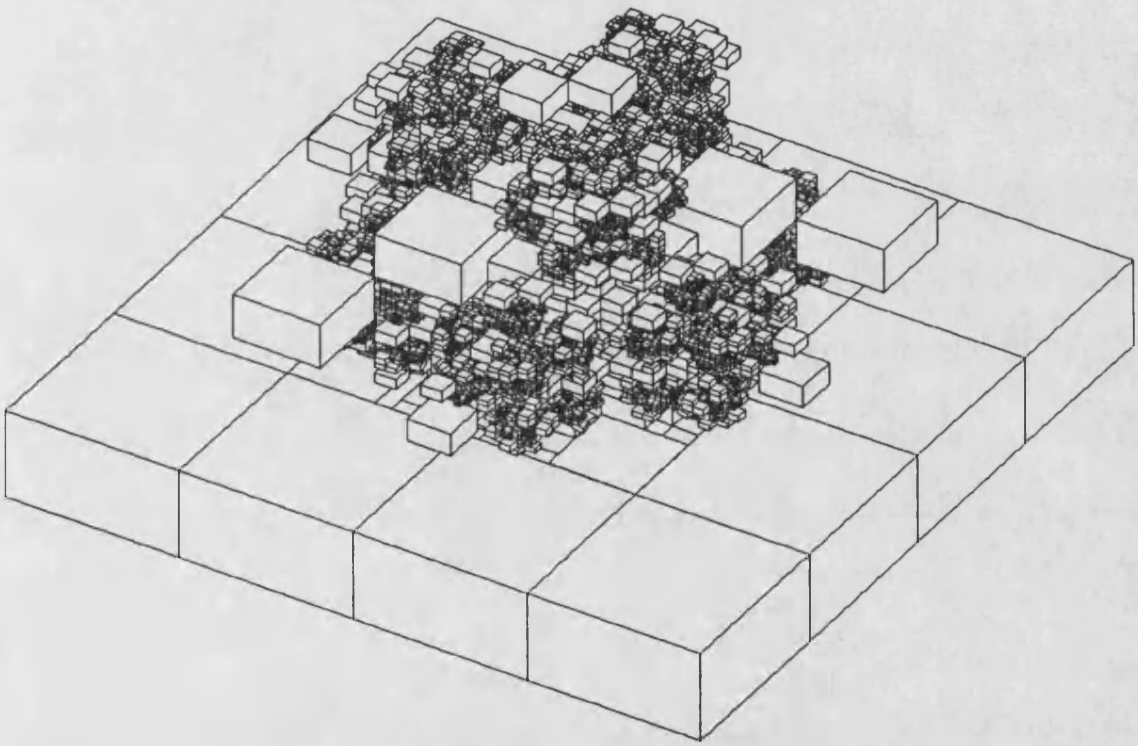


Fig 7.4a

Fig 7.4b: Case Study 2

Experimental Results for Image Synthesis

Case Study 2	5555 Objects		433169 Rays Traced	
Acceleration Technique	Cost			
	Time (days:hours:minutes:seconds) *(% Column Maximum)*			Storage (Kbytes) *(% Col. Max.)*
	Construction	Image Synthesis	Construction + Synthesis	Disk Space
Naive *(Estimate)*	*n/a*	22:09:00:00 *100*	22:09:00:00 *100*	*n/a*
Bounding Volume Hierarchy	40:19 *100*	1:05:34 *0.2*	1:45:53 *0.33*	245 *22*
Grid Partition *Depth 6*	10:18 *26*	59:29 *0.18*	1:09:47 *0.22*	1096 *100*
Octtree *Simplicity 0*† *Depth 6*	10:10 *25*	56:55 *0.18*	1:07:05 *0.21*	250 *23*

† A voxel is decomposed only if its heterogenous list length exceeds 0

Case Study 2: The 5555 Object Scene

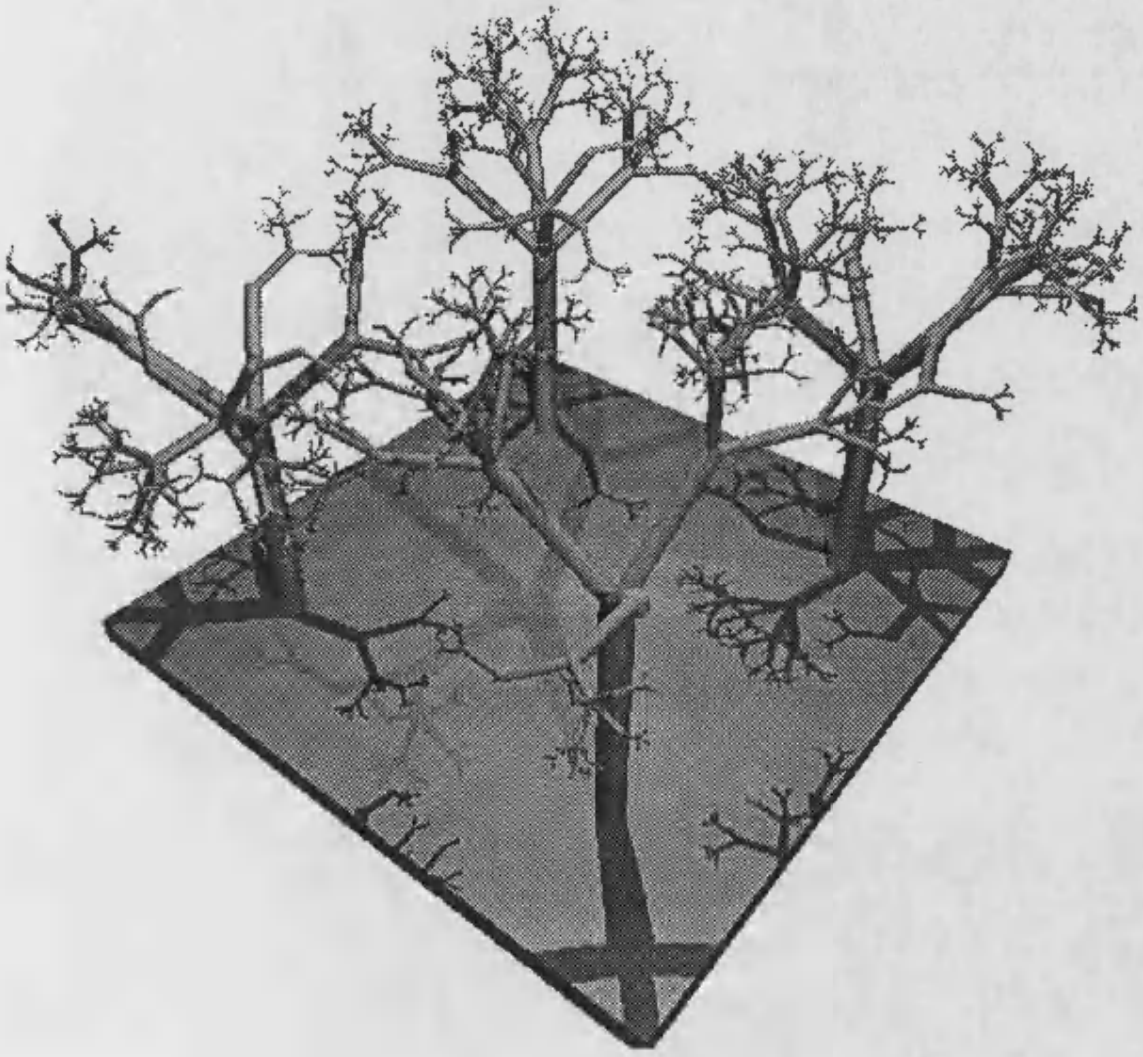


Fig 7.4b

Case Study 2: A Perspective View of the Bounding Volume Hierarchy

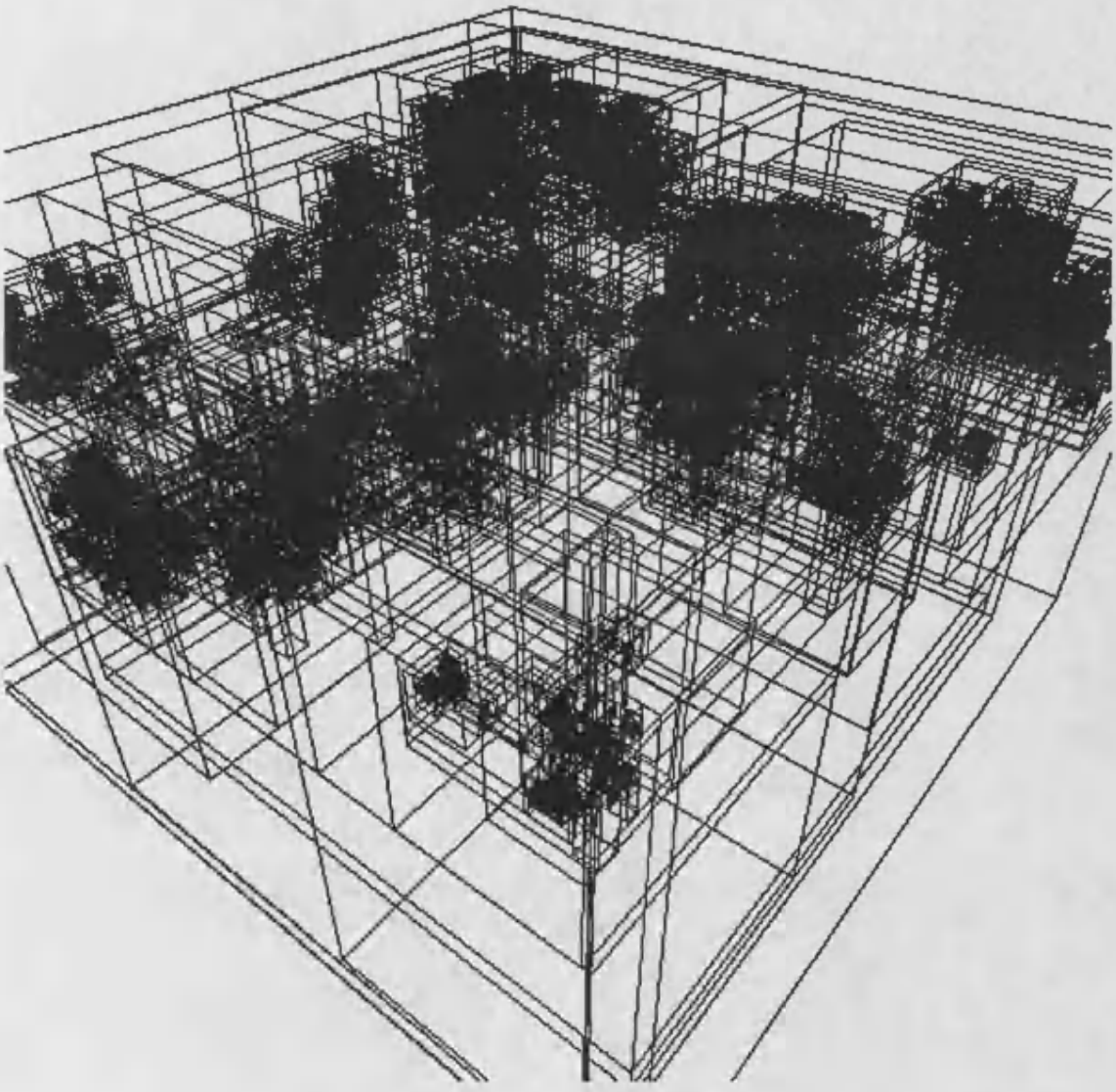


Fig 7.4b

Case Study 2: A Parallel Projection of Non Empty Cells in the Grid Partition or Non Empty Voxels in the Octtree Decomposition

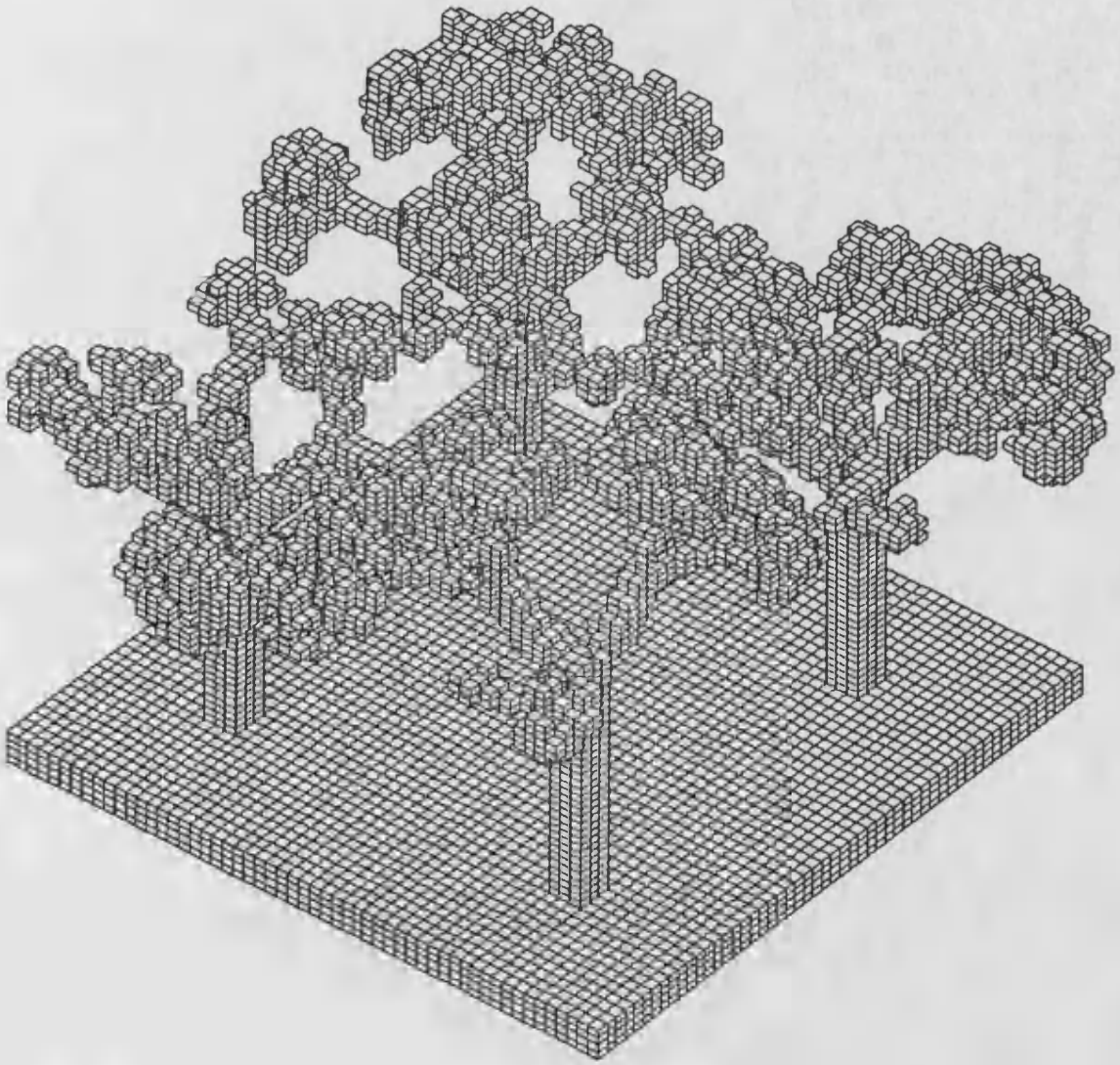


Fig 7.4b

Fig 7.4c: Case Study 3
Experimental Results for Image Synthesis

Case Study 3	320 Objects		376995 Rays Traced	
Acceleration Technique	Cost			
	Time (days:hours:minutes:seconds) *(% Column Maximum)*			Storage (Kbytes) *(% Col. Max.)*
	Construction	Image Synthesis	Construction + Synthesis	Disk Space
Naive *(Estimate)*	*n/a*	1:23:00:00 *100*	1:23:00:00 *100*	*n/a*
Bounding Volume Hierarchy	9 *6.5*	51:01 *1.8*	51:10 *1.8*	14 *1.3*
Grid Partition *Depth 6*	2:19 *100*	24:31 *0.87*	26:50 *0.95*	1061 *100*
Octtree *Simplicity 0*† *Depth 6*	2:19 *100*	27:48 *0.99*	30:07 *1.1*	172 *16*

† A voxel is decomposed only if its heterogenous list length exceeds 0

Case Study 3: The 320 Object Scene

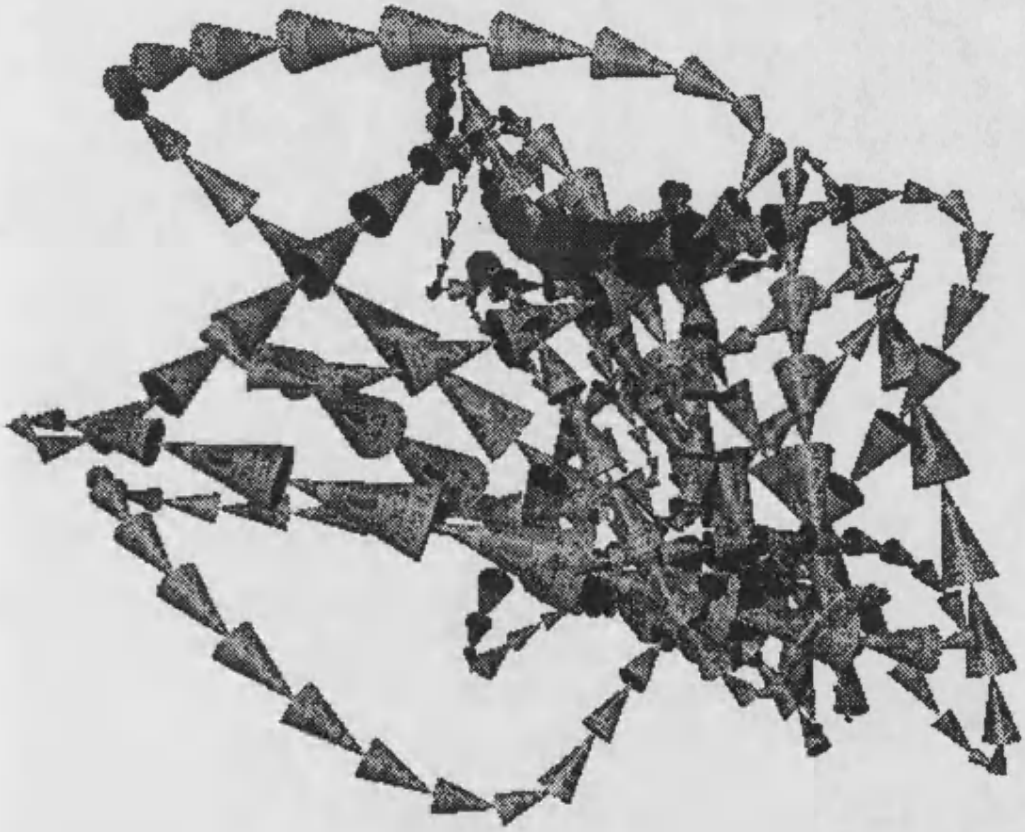


Fig 7.4c

Case Study 3: A Perspective View of the Bounding Volume Hierarchy

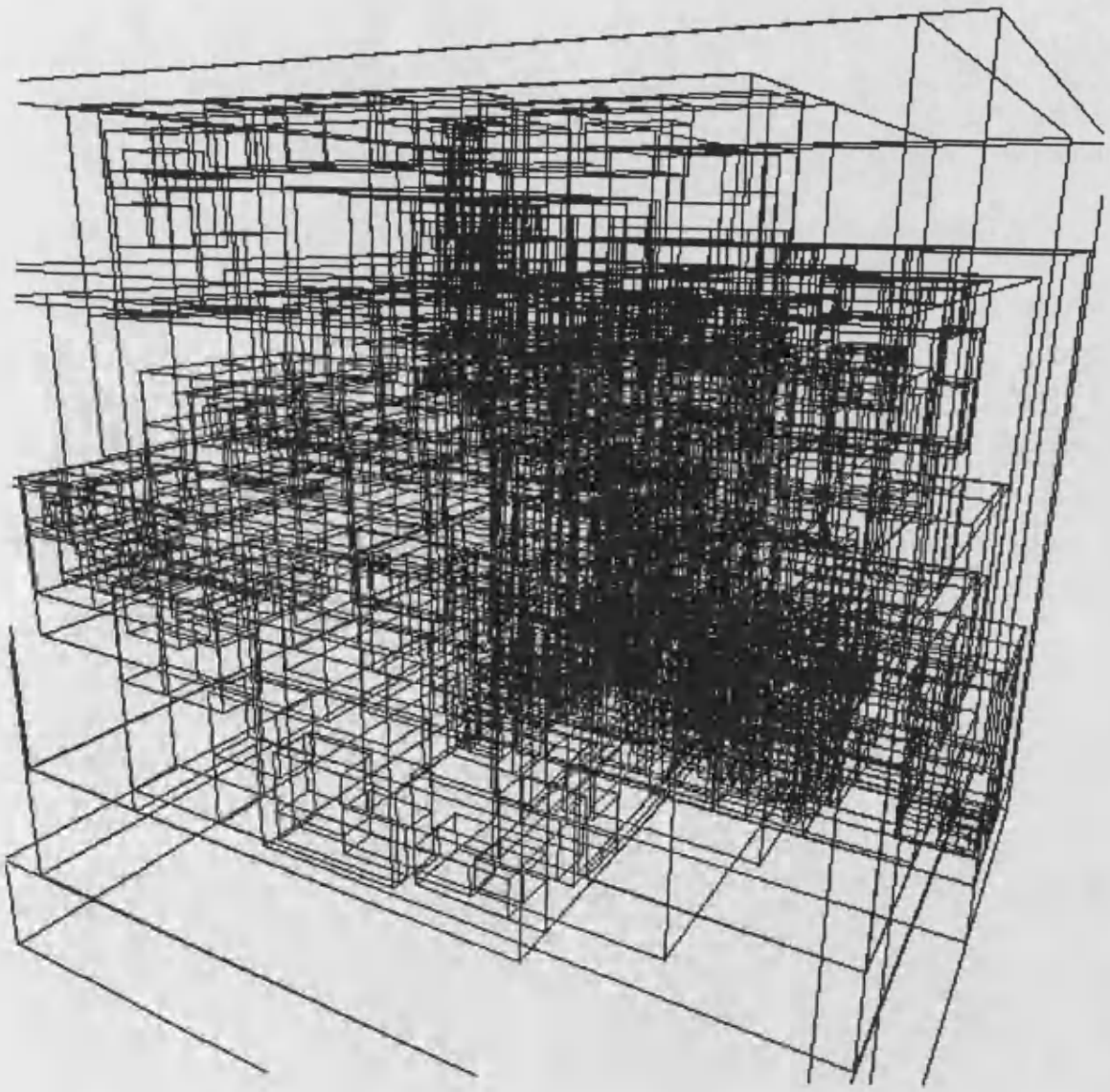


Fig 7.4c

Case Study 3: A Parallel Projection of Non Empty Cells in the Grid Partition or Non Empty Voxels in the Octtree Decomposition

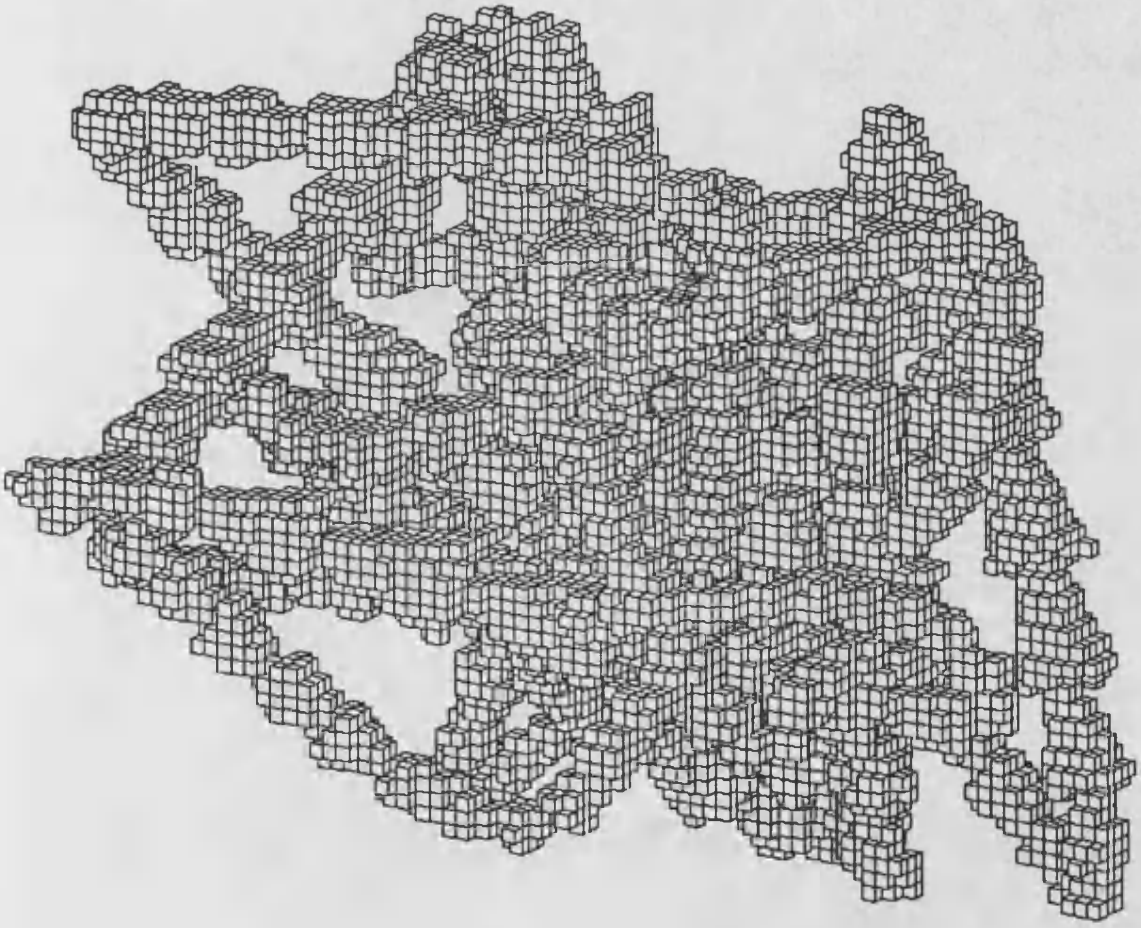


Fig 7.4c

Fig 7.4d: Case Study 4

Experimental Results for Image Synthesis

Case Study 4	106 Objects		676894 Rays Traced	
Acceleration Technique	Cost			
	Time (days:hours:minutes:seconds) *(% Column Maximum)*			Storage (Kbytes) *(% Col. Max.)*
	Construction	Image Synthesis	Construction + Synthesis	Disk Space
Naive *(Estimate)*	*n/a*	1:22:00:00 *100*	1:22:00:00 *100*	*n/a*
Bounding Volume Hierarchy	1 *0.18*	2:06:18 *4.6*	2:06:19 *4.6*	9 *0.64*
Grid Partition *Depth 6*	2:18 *25*	1:26:28 *3.1*	1:28:46 *3.2*	1053 *75*
Octtree *Simplicity 0*† *Depth 7*	9:13 *100*	1:35:13 *3.4*	1:44:26 *3.8*	1410 *100*

† A voxel is decomposed only if its heterogenous list length exceeds 0

Case Study 4: The 106 Object Scene

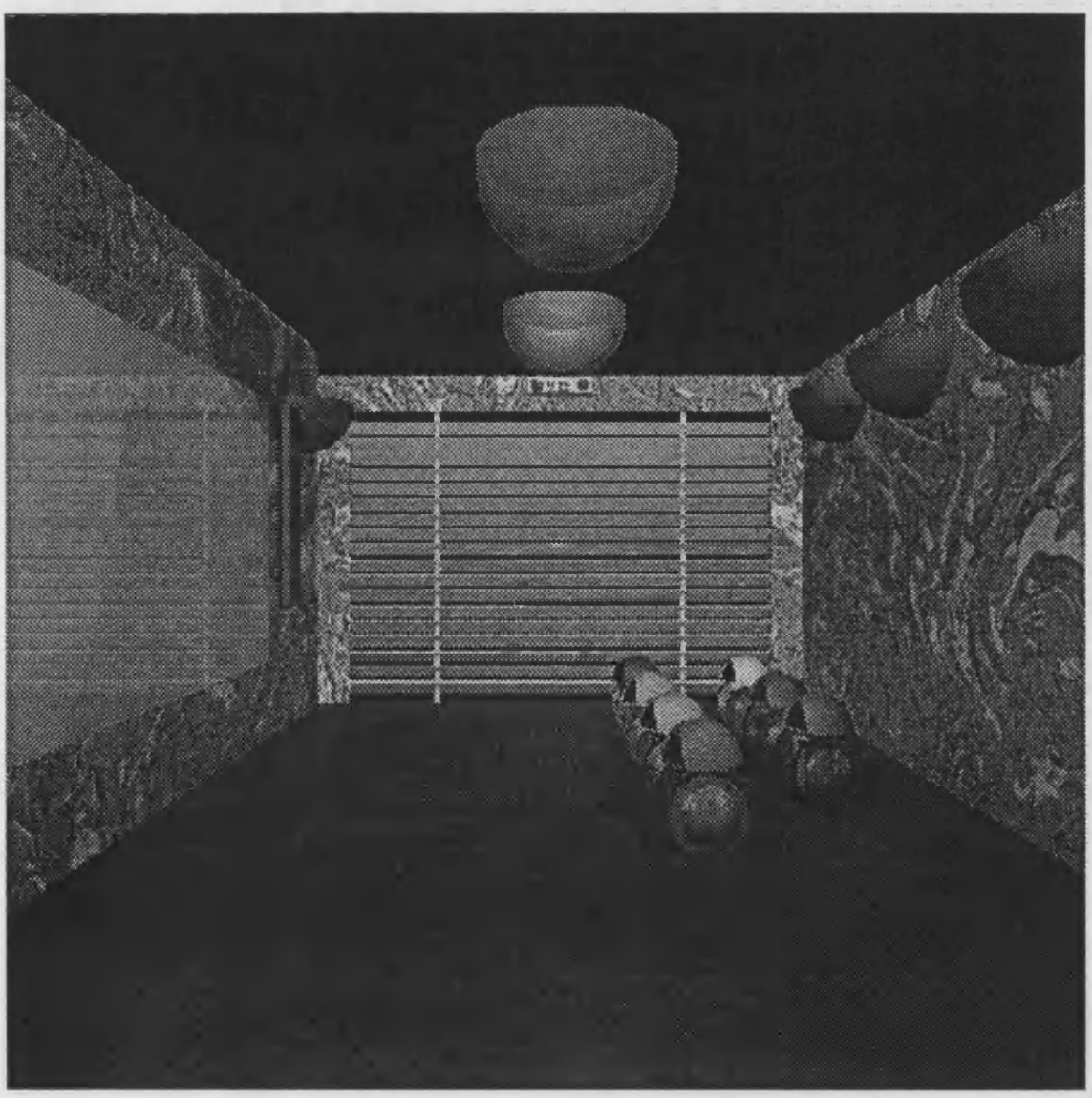


Fig 7.4d

Case Study 4: A Perspective View of the Bounding Volume Hierarchy

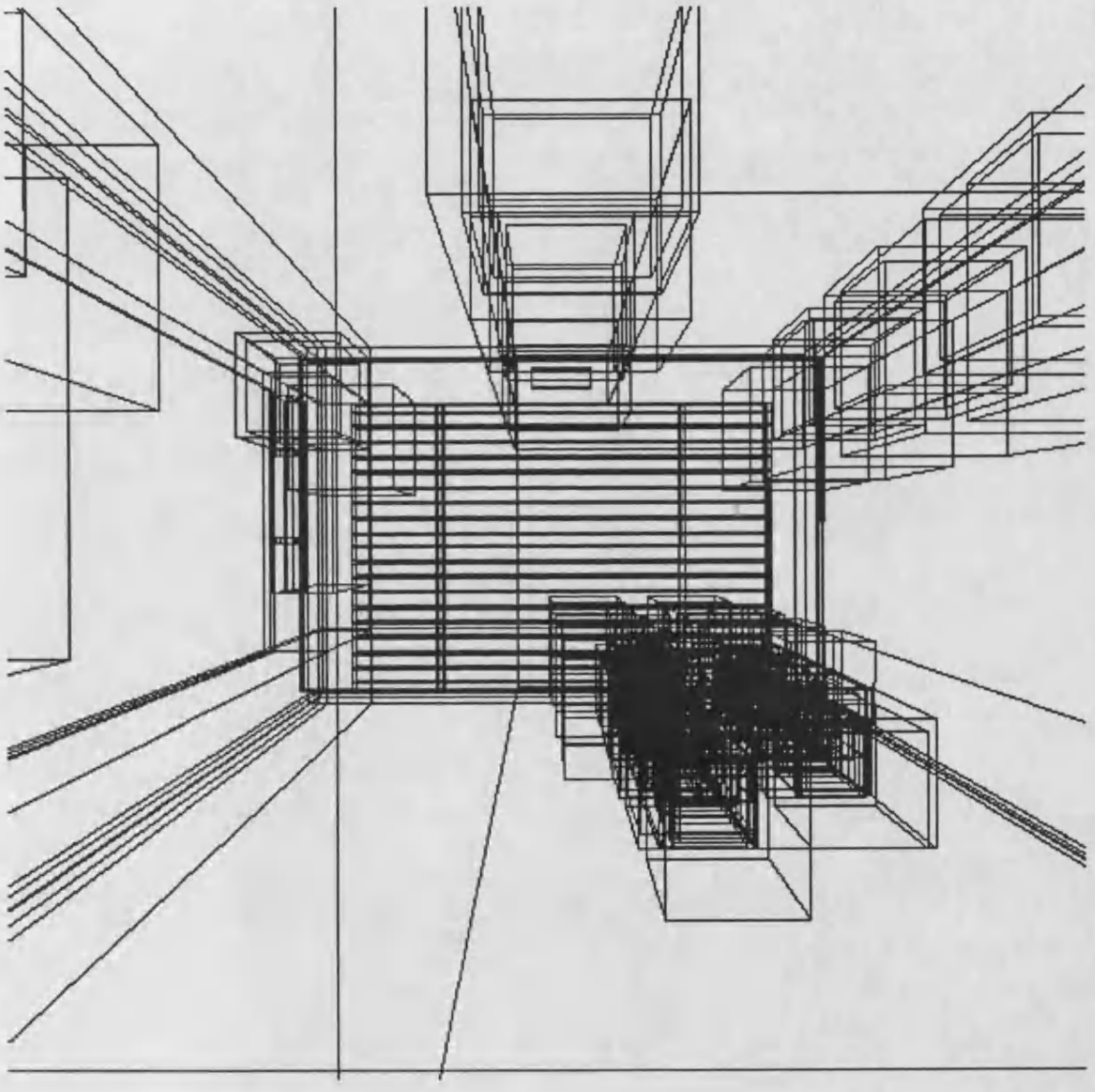


Fig 7.4d

Case Study 4: A Parallel Projection of Non Empty Cells in the Grid Partition

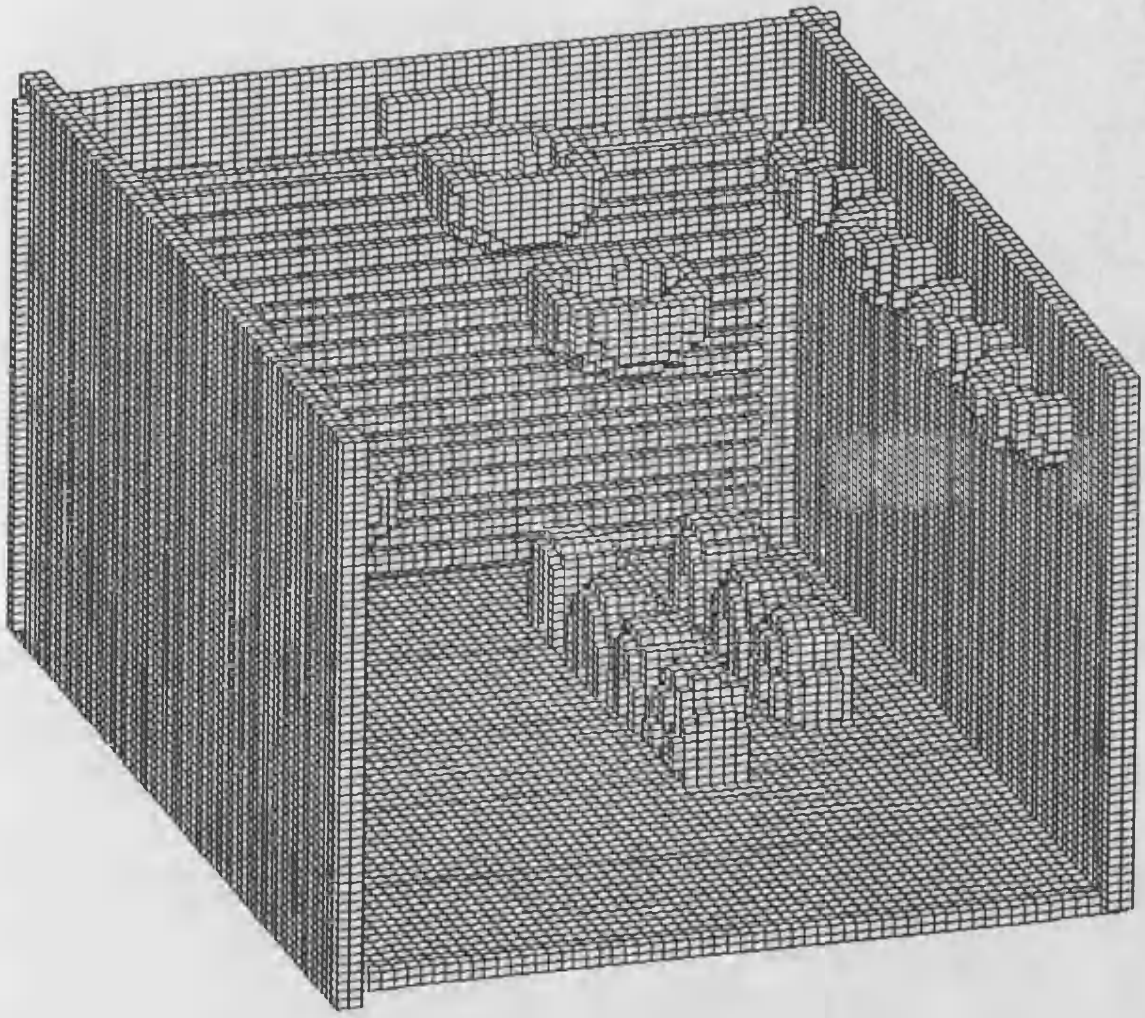


Fig 7.4d

Case Study 4: A Parallel Projection of Non Empty Voxels in the Octtree Decomposition

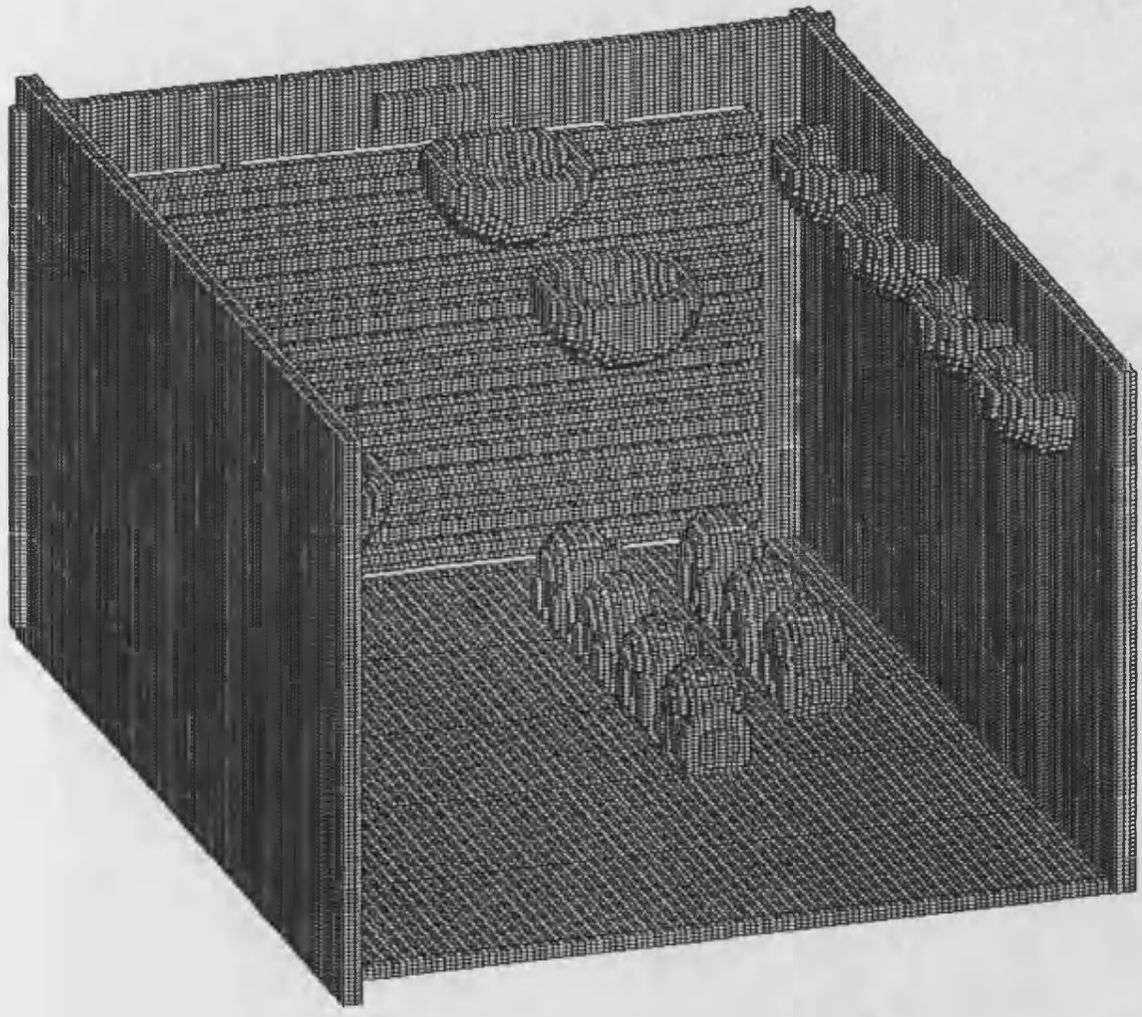


Fig 7.4d

a 2-D spectrum of possible octtrees, parameterised by the twin termination criteria of a voxel's maximum permitted depth and 'simplicity' or maximum permitted heterogeneous list length. The grid partition is constructed to a depth of six for each test piece, requiring a one Mega-byte cell array. This provides a reasonable degree of decomposition. Each further depth of decomposition would require eight times more memory. This could not be feasibly allocated on the Orion as exclusive primary storage to the extended image synthesis. However, the octtree decomposition is more flexible in its demands on memory. A suitable octtree is tailored for each test piece to provide an adequate degree of local scene simplification within reasonable storage space. The termination criteria are given for each test piece.

7.5. An Analysis of the Experimental Results

Various significant observations may be drawn from the experimental results [Fig 7.4a-d].

7.5.1. The Acceleration of Image Synthesis

The most fundamental observation is that each scene decomposition succeeds in a dramatic acceleration over naive synthesis for every test scene. Estimated total times for naive ray tracing range from just under two days to over *two months*. Times for the scene decompositions range from about half an hour to four hours, and from 0.14% to 4.6% of their estimated naive counterparts.

7.5.2. The Influence of Object Count on Acceleration

The degree of acceleration achieved clearly increases with object count. Relative times for the scene decompositions are between 3.2-4.6% of estimates under naive synthesis for the 106 object scene [Fig 7.4d], 0.95-1.8% for the 320 object scene [Fig 7.4c], 0.21-0.33% for the 5555 object scene [Fig 7.4b] and 0.14-0.27% for the 7832 object scene [Fig 7.4a]. This is to be expected due to the increased opportunity of avoiding expensive object intersections when solving the scene model. A comparison of the experimental results between acceleration techniques reveals further interesting observations.

7.5.3. A Comparison of Total Times

Consider the total construction and synthesis times. Total times for the bounding volume hierarchy always exceed those for the grid partition and octtree. The grid partition achieves the fastest times for scenes of low object count, but is progressively superseded by the octtree for more complex scenes. Considering the test scenes by increasing object count again, total times for the grid partition decrease in the sequence 3.2%, 0.95%, 0.22%, 0.21% of estimates for naive synthesis, and for the octtree in the sequence 3.8%, 1.1%, 0.21%, 0.12%. The superiority of the octtree is to be expected at high object count since more local scene simplification is achieved within a decomposition of bounded size.

7.5.4. A Comparison of Construction Times

Consider the time spent in the construction of these decompositions. Construction times for the bounding volume hierarchy grow rapidly with object count, from one second to over an hour and a half. This is to be expected due to the complexity of allowing for dependencies between many bounds in the construction of an efficient hierarchy. Construction times for the grid partition and octtree are more stable, with the former varying between about two and ten minutes, and the latter between two and twenty minutes. These times are clearly influenced by factors other than object count. The construction of the grid partition takes over ten minutes for the 5555 object scene [Fig 7.4b] but less than five minutes for the 7832 object scene [Fig 7.4a]. Similarly, octtree construction takes over nine minutes for the 106 object scene [Fig 7.4d] compared to under three minutes for the 320 object scene [Fig 7.4c].

7.5.5. A Comparison of Synthesis Times

Consider the image synthesis times when tracing rays through these decompositions. Suitable choices of octtree decomposition consistently provide superior synthesis times to the bounding volume hierarchy. The less flexible grid partition achieves the same for all but the scene of highest object count [Fig 7.4a]. This is not surprising since the degree of local scene simplification within the fixed depth grid partition may be expected to decrease with

object count. The depth of a memory-intensive grid partition is severly limited by the amount of available storage, unlike the more memory-efficient octtree. The synthesis of an animated fly-by of this scene would be more efficient with the bounding volume hierarchy than the grid partition beyond a critical number of frames, but would always be most efficient with the octtree. The grid partition provides the fastest synthesis for scenes with low object count, but is progressively superseded by the octtree for more complex scenes once more.

7.5.6. A Comparison of Storage Requirements

Finally, consider the storage requirements for these decompositions. The bounding volume hierarchy has the lowest requirement for each test piece, despite taking the greatest construction times for the two scenes of greatest object count. The requirement increases approximately linearly with object count, but exceeds this trend for the scene of lowest object count [Fig 7.4d] where bounds are extensively used within objects. The grid partition's requirement constitutes a one Mega-byte or 1024K cell array and a variable amount of storage for the distinct heterogeneous lists. The former is consistently dominant and the total requirement increases marginally with object count from 1053K to 1143K. The octtree requires a variable amount of storage both for the Autumnal representation of its structure and the distinct heterogeneous lists. The requirement is less than a quarter of the grid partition's when constructed under identical termination criteria whilst construction and synthesis times are roughly equal [Fig 7.4b,c]. Moreover, the octtree's requirement is far less than the grid partition's would be for deep decompositions [Fig 7.4a,d].

7.6. Predictions for the General Performance of Acceleration Techniques

Many factors outside the scope of a scene decomposition can influence synthesis times for a given scene, as previously remarked [Section 7.3]. This complicates any attempt to extrapolate data measured from case studies to a general scene. To be at all reliable, this would require the synthesis of more 'bench mark' case studies than feasible.

The number of rays traced and the computational rate of the host machine have a particularly significant influence on synthesis times. Traditionally, researchers have presented experimental data for their *own* case study images on their *own* hardware. These case studies vary greatly in nature, and a wide range of machines of disparate power are present throughout the computer graphics research community. Some researchers have synthesised images with a 4096 processor array [Williams et al;1986]. Any attempt to draw meaningful conclusions from comparisons in synthesis times is complicated when many such factors vary. A data base of test scenes has been proposed [Haines;1987]. However, this has not yet entered into wide use and is hence of limited application in comparing the acceleration techniques proposed by different researchers. Moreover, the data base addresses only simple scene and view models without CSG, stretch and shear deformations, or texturing. Whilst the synthesis of such images may be accelerated by abandoning support for such sophistication, such restriction is clearly undesirable.

However, simple models may be constructed to predict the behaviour of the proposed acceleration techniques in response to a variation in just one factor of interest. These predictions may be verified experimentally to indicate which decomposition is preferable in the general case.

7.7. The Influence of Object Count on Performance

A realistic model of a scene may include thousands of objects to provide detail. The influence of object count on the performance of image synthesis is of major importance to many applications. A table of predictions is given for the influence of object count under the assumption that all other determining factors remain constant [Fig 7.7a]. The derivation of this table and other associated assumptions are given below, with some experimental verification of the predictions.

Fig 7.7a: Predictions of the Influence of Object Count on the Performance of Image Synthesis

General Case	'N' Objects	Other Factors Constant	
Acceleration Technique	Cost		
	Time		Storage
	Construction	Image Synthesis	Disk Space
Naive	*n/a*	O(N)	*n/a*
Bounding Volume Hierarchy	$O(N^2)$	O(log(N))	O(N)
Grid Partition	O(N)	O(1)	O(1)†
Octtree	O(N)	O(1)	O(N)‡

† Constant but large

‡ But no more than $\frac{1}{7}$ greater than for the grid partition, and generally less.

7.7.1. Construction Times

The construction of the Huffman derived bounding volume hierarchy repeatedly identifies the optimal pair of bounds to merge until just one bound remains. The identification of each successive pair avoids an exhaustive search where possible but may be expected to consider some fraction of the active bounds. Moreover, the exhaustive search remains the catch-all last resort for several tasks. The identification of each pair may be expected to incur costs which are linear in bound count, giving a total construction cost which is quadratic in object count. This complexity arises from allowing for dependencies between bounds.

Graphs are given for construction times measured experimentally from scenes containing a tree built from progressively more objects with fractal techniques [Fig 7.7.1a]. The plot of the logarithm of construction time against the logarithm of object count indicates a linear relationship of slope ≈ 2. Construction times are thereby verified to be in proportion to object count raised to this power. Tests for other scenes yield similar results.

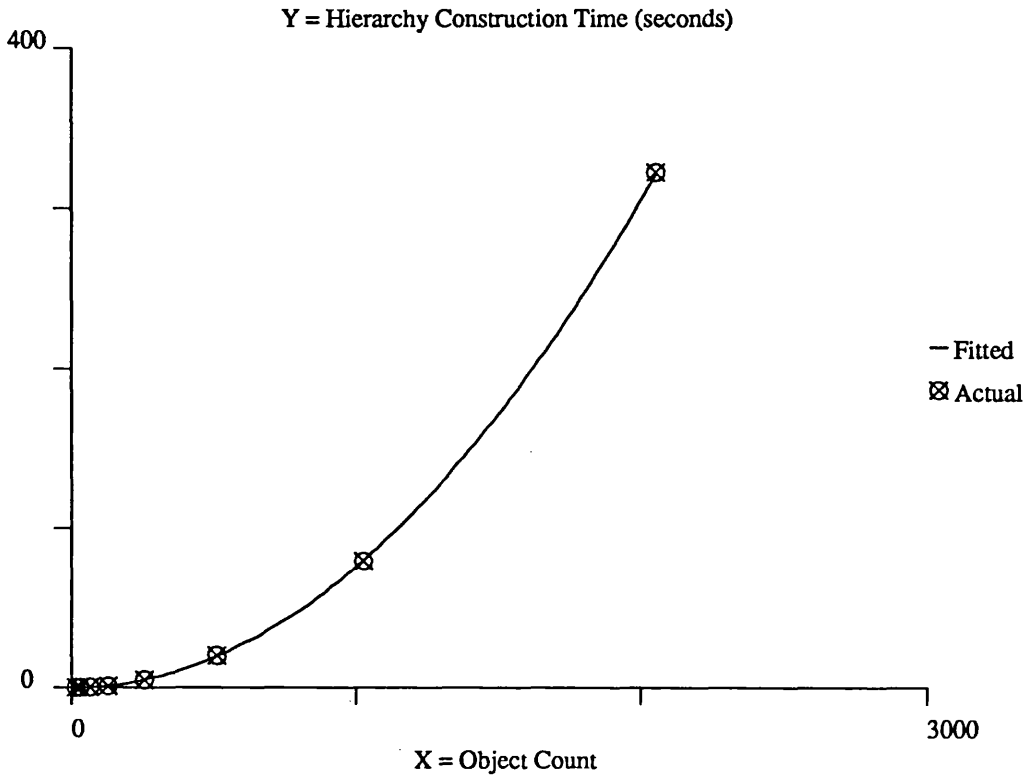
Whilst this rapid growth is undesirable, construction times were always dominated by the savings gained during the synthesis of all the case studies [Fig 7.4a-d].

The grid partition is generated with an octtree decomposition whose simplicity criterion for termination demands that a voxel has an empty heterogeneous list. Assuming that the octtree is generated under similar termination criteria, their construction times may be expected to be approximately equal.

Each object is considered independently during the construction of an octtree decomposition. There is no dependency on any other object. The time required to allow for an object will depend on the number of leaf voxels containing a section of its surface. This depends on the object's surface area which is assumed to be constant. Construction times are therefore predicted to increase linearly with object count.

A graph is given for construction times measured experimentally from scenes containing progressively more spheres of equal size, randomly dispersed within a given box [Fig 7.7.1b]. For each scene an octtree is constructed recursively until the current voxel is at

Fig 7.7.1a: Hierarchy Construction Time is found to be Quadratic in Object Count



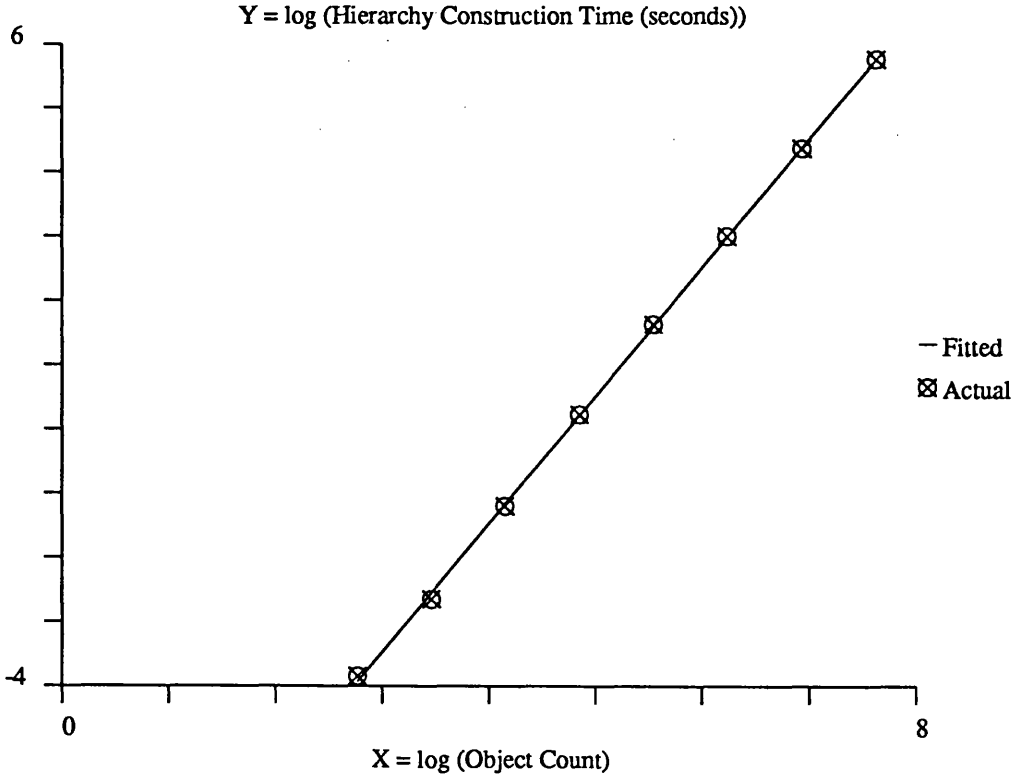


Fig 7.7.1a

Fig 7.7.1b: Octtree Construction Time is found to be Linear in Object Count

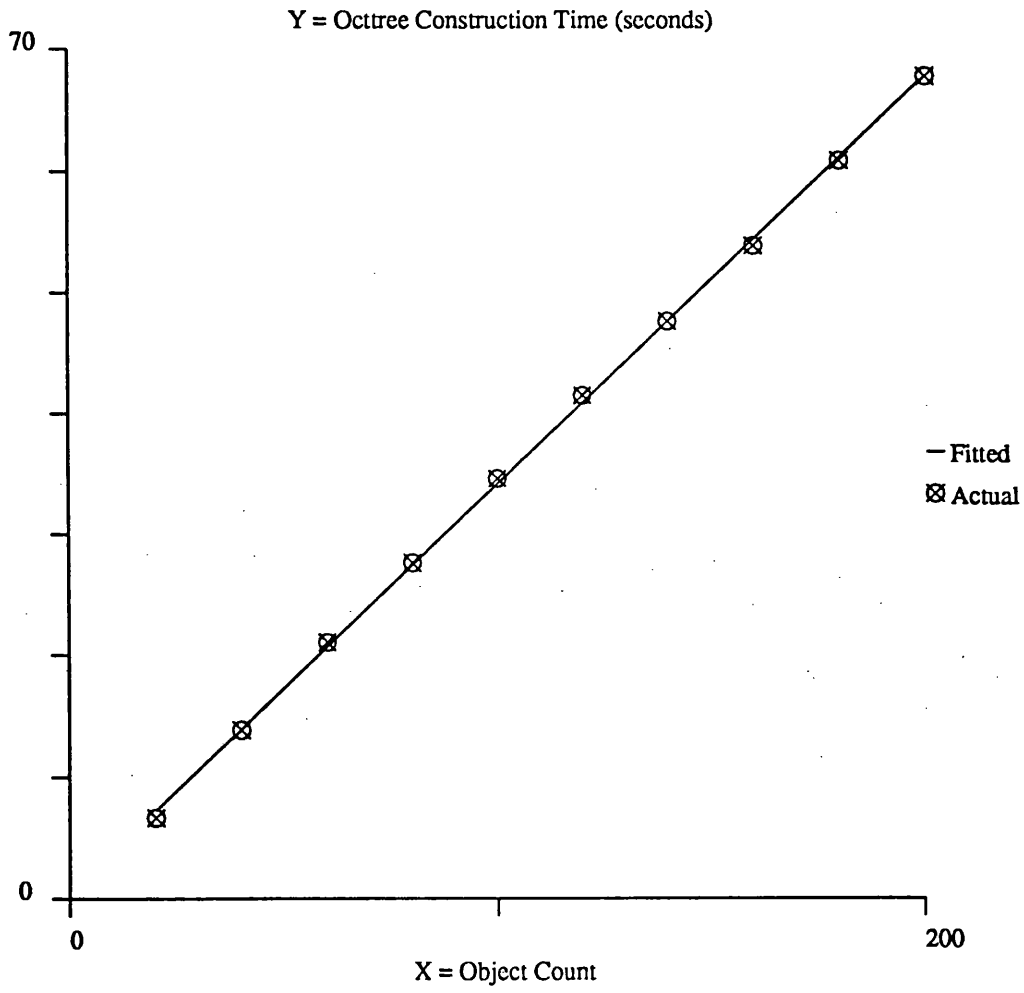


Fig 7.7.1b

depth six or has no heterogeneous objects. The plot of construction time against object count verifies the predicted linear relationship.

7.7.2. Image Synthesis Times

Naive ray tracing solves the scene model for a given ray with an exhaustive object search. This is known to result in synthesis times which are linear in object count [Kay,Kajiya;1986]. This high tariff is the very motivation for scene decomposition.

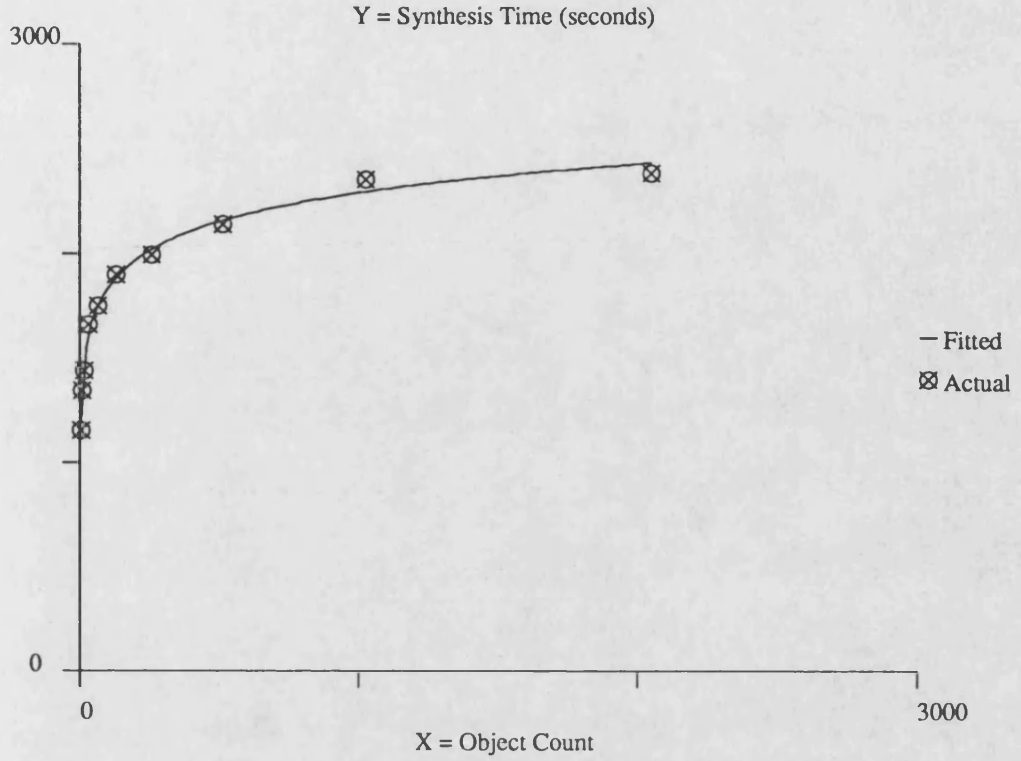
Each node query within a bounding volume hierarchy provides the opportunity to reject an entire branch of objects from the scene model's solution for a given ray. Since the depth of such a hierarchy increases only logarithmically with object count, synthesis times may be expected to behave in the same manner.

Graphs are given for synthesis times measured experimentally from the sequence of scenes described previously containing a tree built from progressively more objects [Fig 7.7.2a]. Each image is synthesised at 512×512 pixels. The plot of synthesis time against the logarithm of object count indicates an approximately linear relationship [Fig 7.7.2a]. Synthesis times are thereby verified to be linear in the logarithm of object count.

This slow growth in synthesis times is preferable by far to the linear growth of naive ray tracing, but will never level out. Many insignificant objects may be rejected from the scene model's solution by a single bound query, but only after this query has been made. Since insignificant objects are considered before being rejected, synthesis times may be expected to keep increasing with object count in general. Moreover, the time taken for the Huffman-derived construction of such a hierarchy has been shown to grow rapidly with object count.

Insignificant objects need never even be considered in the navigation of a grid partition or octtree. Only heterogeneous objects within a local region of the scene come under scrutiny at any stage. The solution of the scene model for a given ray generally constitutes an efficient navigation through empty local regions until a region with some heterogeneous objects is reached. Assuming the degree of local scene simplification is sufficient, the number of heterogeneous objects in this region will not only be low but also *independent* of

Fig 7.7.2a: Image Synthesis Time with the Hierarchy is found to be Logarithmic in Object Count



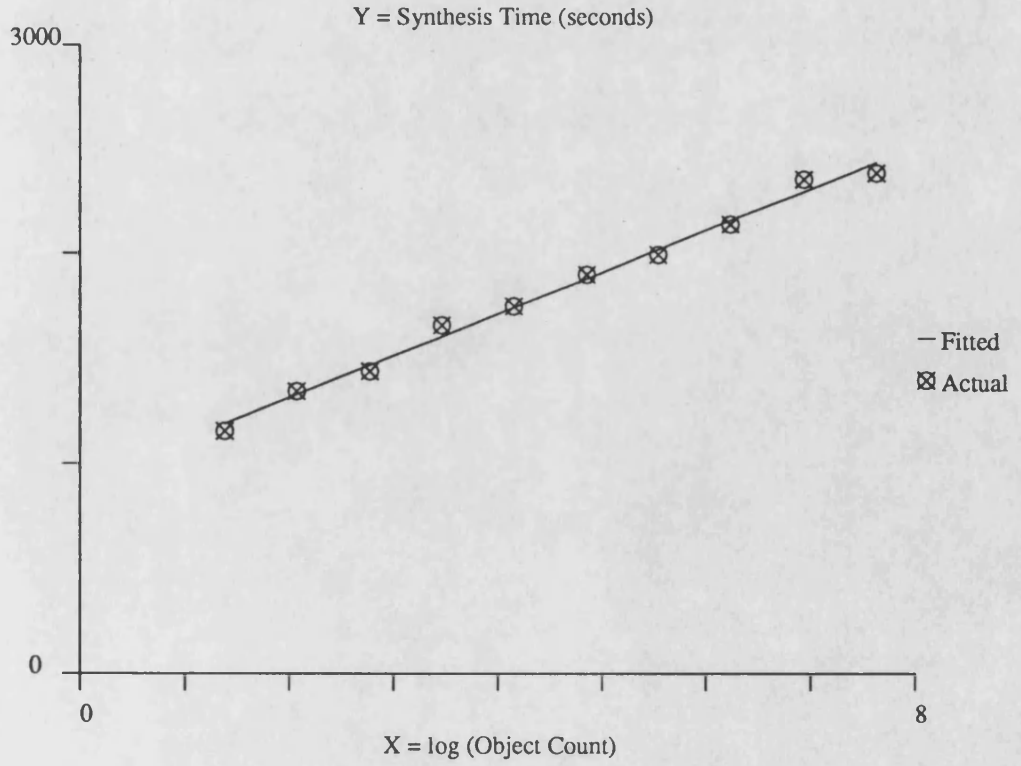


Fig 7.7.2a

the total object count in the scene. Moreover, the scene model solution will often be solved within this region, and the task is then complete. The number of heterogeneous objects in other regions of the scene is of no importance. Synthesis times may therefore be expected to be *independent* of object count. Indeed, experimental results for ARTS [Fujimoto et al;1986] indicate that in exploiting the grid partition "*when the number of objects is very large, ray tracing - despite its reputation for inefficiency - actually becomes faster than other [incremental scan line] methods*".

Synthesis times have been measured experimentally for two scenes with equivalent octtree decompositions. One contains a single torus whilst eighty tori are distributed around a spline path in the other. The pixel coverage of the single torus is made approximately equal to that of the eighty tori to prevent any difference in costs incurred through the spawning of illumination rays, shading calculations and so on during synthesis. Each scene is decomposed with an octtree until the current voxel is at depth seven or has an empty heterogeneous list, and an image is synthesised at 512 × 512 pixels. The synthesis of the single torus image [Fig 7.7.2b] took just over 33 minutes and traced just over 470 thousand rays, whilst the synthesis of the eighty tori image [Fig 7.7.2c] took just over 31 minutes and traced just under 470 thousand rays. Synthesis times are thereby verified to be independent of object count in this case.

Grid partitions of these scenes to depth six proved insufficient to make synthesis times independent of object count. The synthesis of the single torus image took several minutes less than that of the eighty tori image. Synthesis times may be expected to become independent of object count for grid partitions of greater depth, but experimental verification of this has been precluded by limitations in memory. However, experimental results for ARTS [Fujimoto et al;1986] have verified that given sufficient depth, grid partitions do make synthesis times independent of object count.

The prediction of synthesis times being independent of object count is most encouraging. Absolute synthesis times then depend on factors which may be addressed separately, such as the computational power of the host machine. The on-going increase in hardware

Fig 7.7.2b: A Single Torus Scene
Synthesis took 33 minutes with an
Octtree Decomposition to Depth 7, Simplicity 0

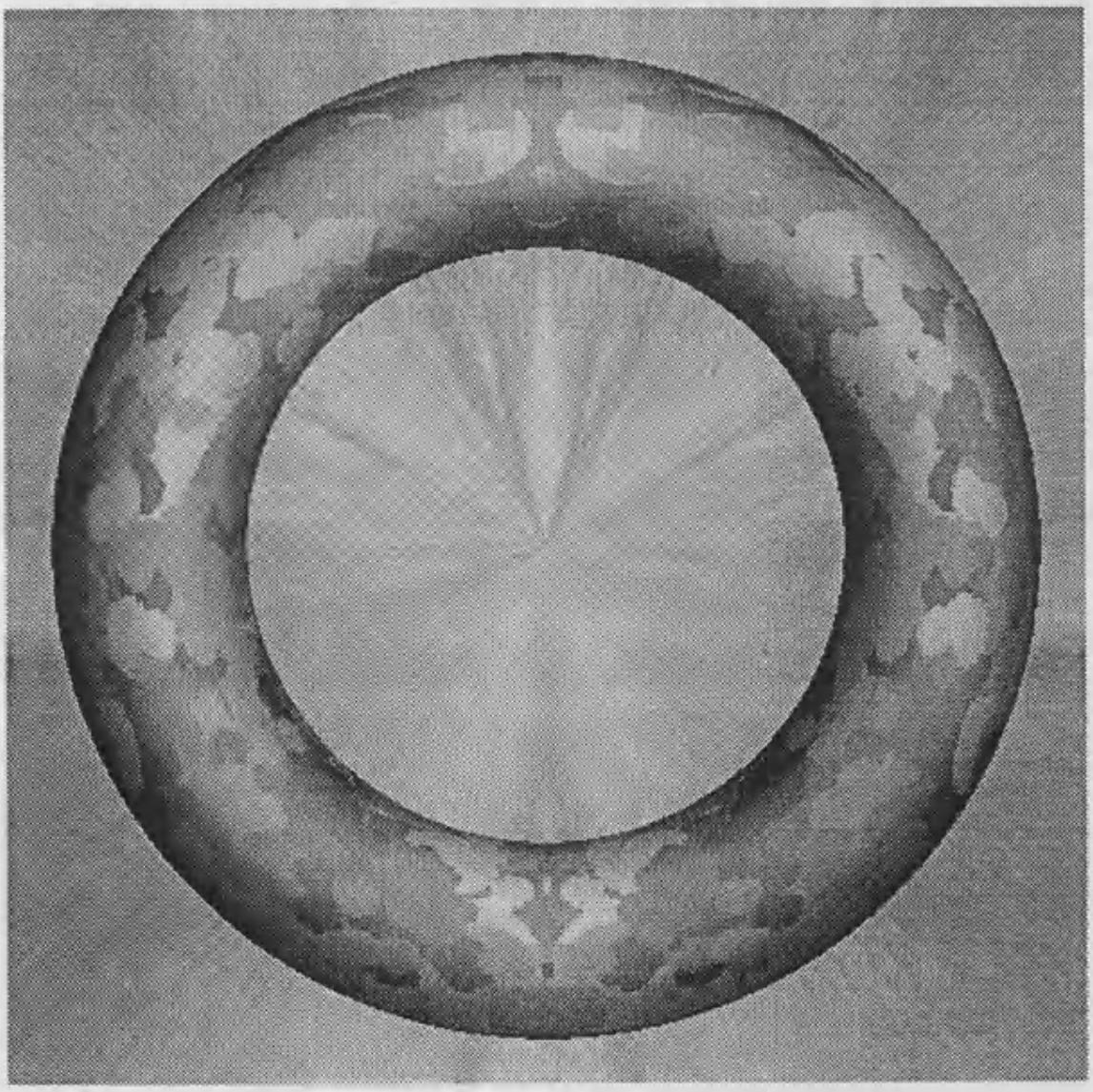
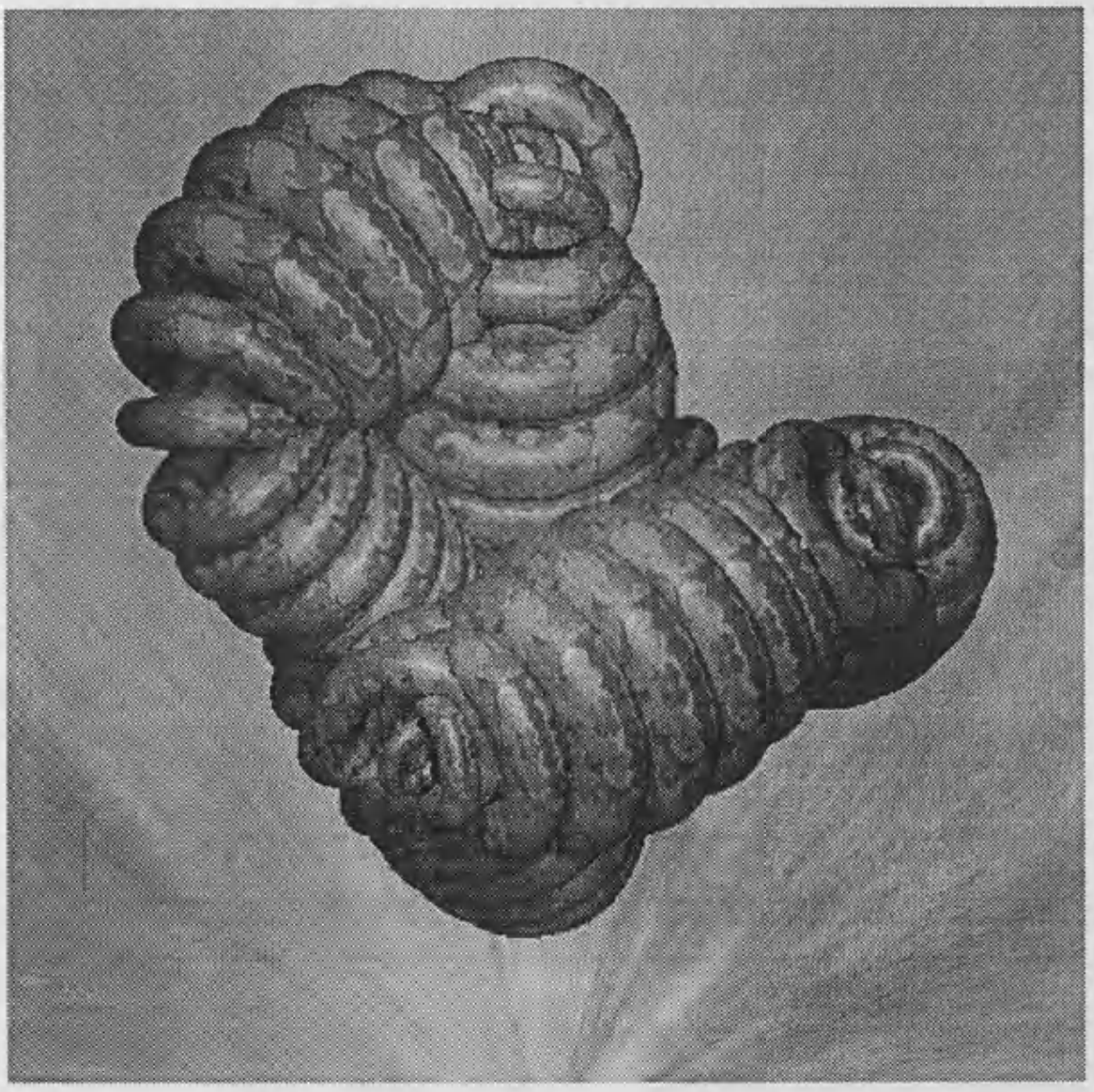


Fig 7.7.2c: An Eighty Tori Scene
Synthesis took 31 minutes with an
Octtree Decomposition to Depth 7, Simplicity 0



technology presents the possibility of real time image synthesis. The advent of parallel processing systems has made this prospect particularly feasible since ray tracing is highly parallelisable [Section 3.4].

7.7.3. Storage Requirement

The Huffman-derived bounding volume hierarchy is built from given leaf bounds as a binary tree. Each bound requires uniform storage, and the total count is one less than twice the leaf count. The storage requirement of the bounding volume hierarchy therefore increases linearly with object count.

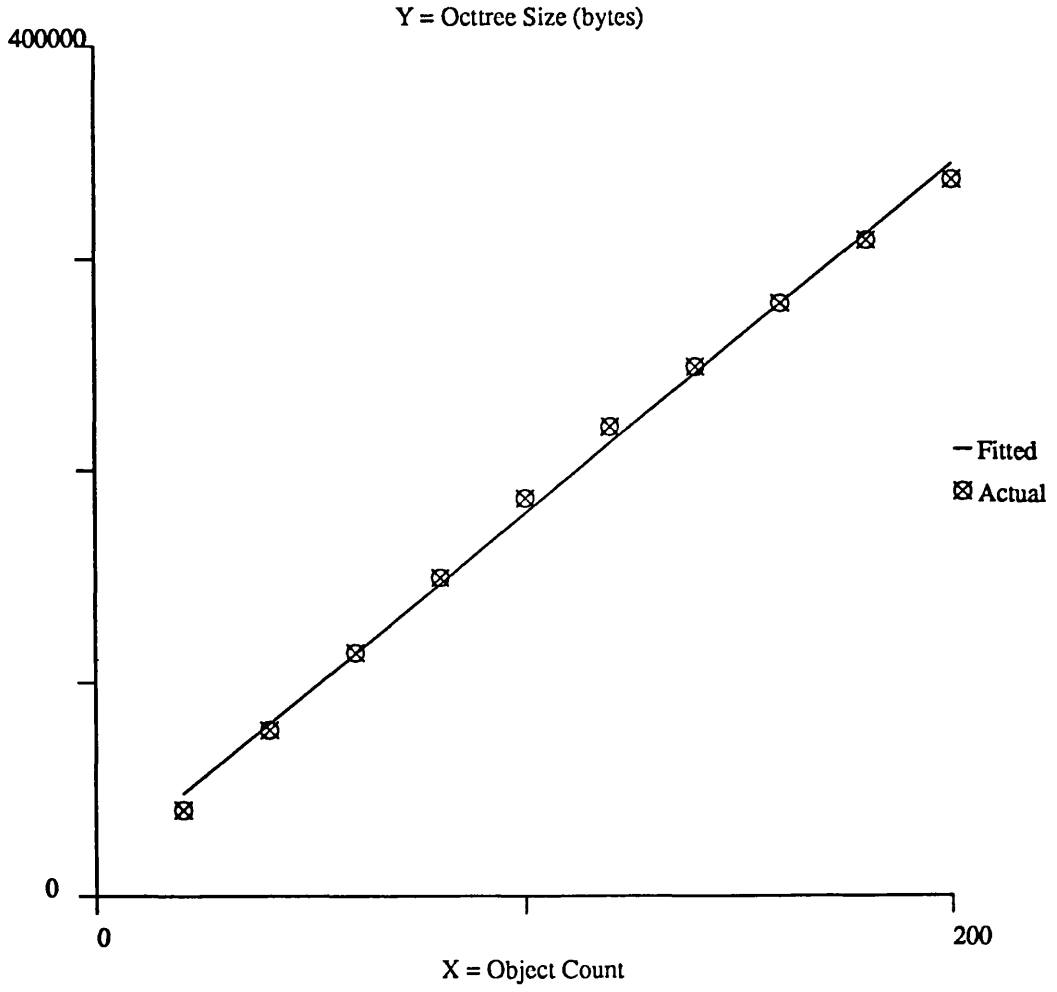
The storage requirement of a grid partition constitutes a cell array of fixed size for given decomposition depth and a variable amount of storage for the cells' heterogeneous object lists. The former generally dominates so the storage requirement of the grid partition is approximately independent of object count [Section 7.5.6].

The storage requirement of the octtree constitutes variable amounts of storage both for the Autumnal representation of its structure and the leaf voxels' heterogeneous object lists. The former dominates in general and is proportional to the octtree's internal node count. When decomposing any voxel with a non empty heterogeneous object list, the octtree complexity theory predicts that the storage requirement will increase linearly with the surface area of the scene objects [Section 6.6]. Assuming objects to be of approximately equal surface area, the storage requirement of the octtree will increase linearly with object count.

A graph is given for the storage requirement of an octtree measured experimentally from the sequence of scenes described earlier containing progressively more spheres of equal size, randomly dispersed within a given box [Fig 7.7.3a]. Each scene is decomposed until the current voxel is at depth six or has an empty heterogeneous object list. The plot of storage requirement against object count verifies the predicted linear relationship.

The storage requirement of an octtree constructed under a less demanding simplicity criterion would be smaller than under this strict criterion [Section 6.6]. In general, the

Fig 7.7.3a: The Octtree's Storage Requirement is found to be Linear in Object Count



octtree's requirement will never be more than a seventh greater than that of a grid partition constructed to the same depth [Section 6.6].

7.8. The Influence of Decomposition Depth on Performance

Whilst most welcome, the prediction of synthesis times being independent of object count for the grid partition and octtree are made under the assumption of adequate local simplification within the scene. This requires a degree of decomposition sufficient to ensure that the number of heterogeneous objects in each local region is both low and independent of the global object count. The influence of the depth of decomposition on storage requirements is of major importance in assessing how realistic this assumption can be.

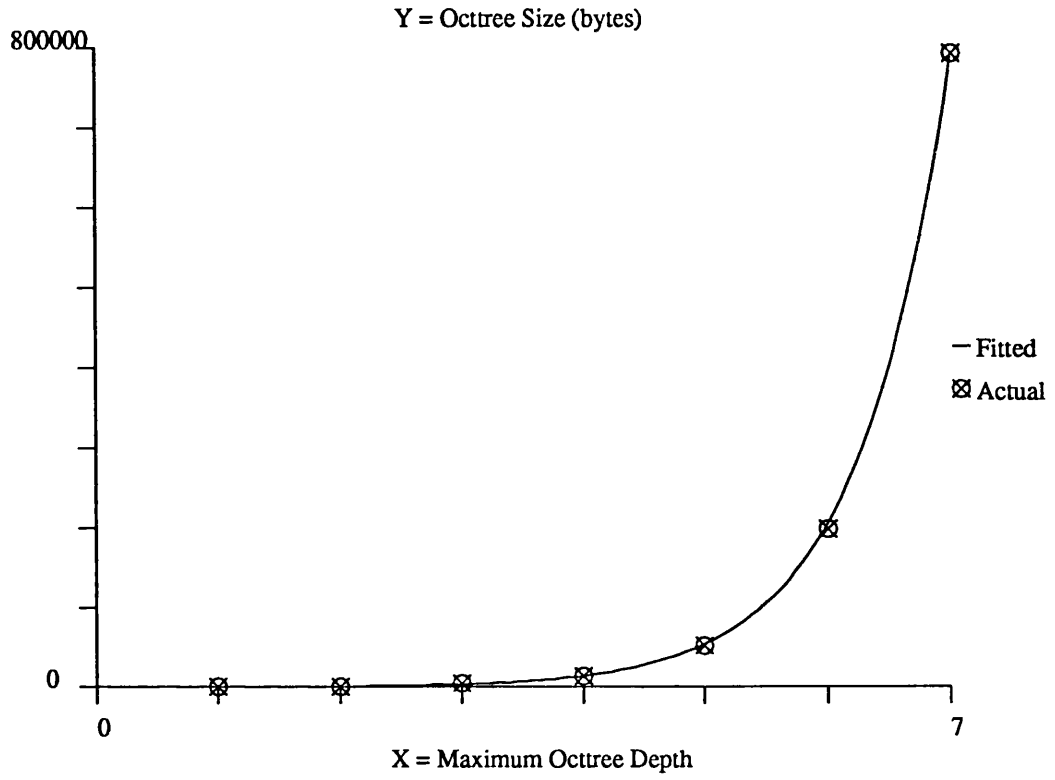
7.8.1. Storage Requirement

The size of the grid partition's cell array increases at an exponential rate of eight with decomposition depth. This rapid growth severely restricts the viable depth of decomposition. The assumption of adequate local scene simplification therefore becomes less realistic for scenes of high object count.

The storage requirement of the octtree however depends not only on the maximum depth of the decomposition but also the nature of the scene and the simplicity criterion for termination. Growth for highly demanding simplicity criteria may still be expected to be exponential in object count but at a lower rate [Section 6.6]. For less demanding simplicity criteria, the storage requirement may be expected to become constant provided objects do not overlap [Section 6.6].

Graphs are given of storage requirements measured experimentally for octtrees constructed to progressively greater depths [Fig 7.8.1.a] for one of the case studies [Fig 7.4d] . Each construction is terminated by simplicity criterion of zero. The plot of the logarithm of storage requirement against decomposition depth indicates a linear relationship of slope ≈ 1.335. Storage requirements therefore grow at exponential rate of $e^{1.335} \approx 3.8$ with respect to object count in this case.

Fig 7.8.1a: The Octtree's Storage Requirement is found to be Exponential in Depth at Rate 3.8~4



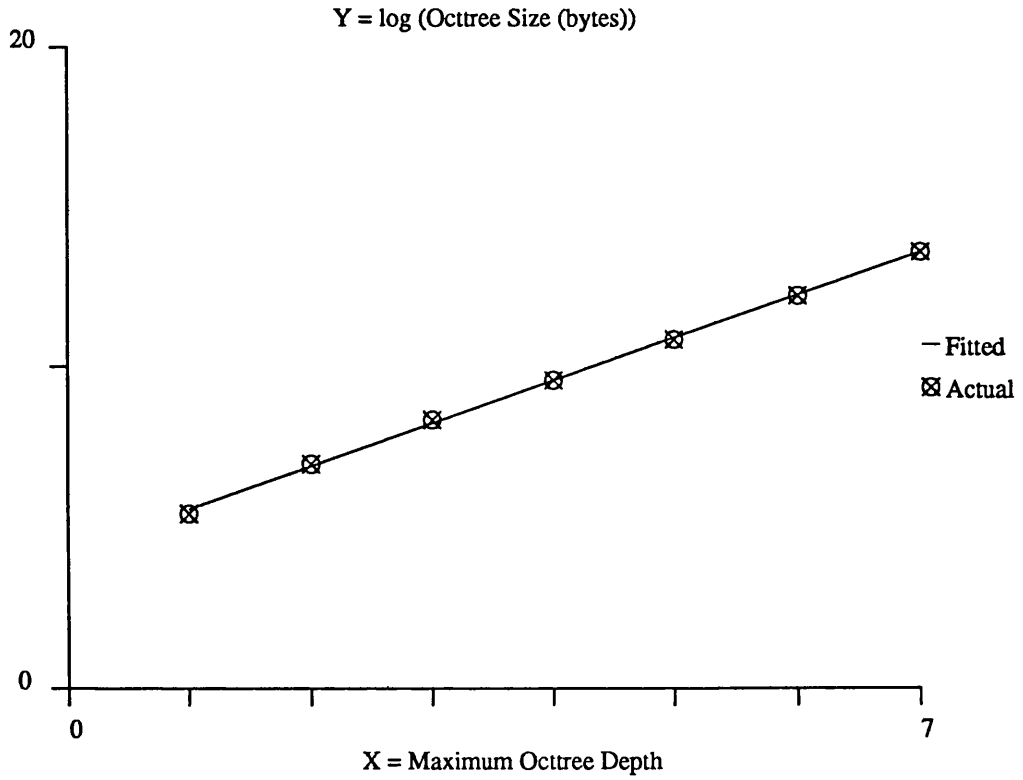


Fig 7.8.1a

7.8.2. Navigation Costs

The local simplification of a scene to a degree ensuring that image synthesis times are independent of object count has been shown to be more easily achieved by an octtree than a grid partition within bounded space. Even if this space restriction is lifted, the smaller octtree may still be expected to incur lower navigation costs for deep decompositions than the grid partition. The significance of navigation costs increases with decomposition depth, since a progressively higher fraction of scene model solution time is spent in navigation rather than calculating object intersections.

Each driving axis step in a ray's navigation of a grid partition traverses up to three grid cells. The total number of cells in a partition increases exponentially with decomposition depth. Navigation costs may therefore be expected to increase in the same manner.

Each voxel traversed in a horizontal step of an octtree contains a number of maximum depth cells equal to eight to the power of the difference between that voxel's depth and the maximum depth. The opportunities for savings in navigational costs over the grid partition clearly increase with decomposition depth. Assuming that the scene model is usually solved inside the first leaf voxel reached with a non-empty heterogeneous list, the number of voxels navigated by each ray may be expected to increase linearly with decomposition depth. Navigation costs may therefore be expected to increase in the same manner.

7.9. Conclusions from the Case Studies and Predictions of Performance

Whilst the bounding volume hierarchy provides a significantly faster image synthesis than naive ray-tracing for the case studies, the other decompositions do better still. Moreover, the hierarchy's construction costs grow rapidly with object count and may be expected to become excessive for more complex scenes than those tested.

The grid partition provides the fastest synthesis for the case studies of lower object count. The limited depth of the grid partition may be expected to provide adequate local simplification in such scenes. Navigation costs are limited at such depths. Up to three cells are traversed in the navigation of each driving axis step, and the whole grid partition

may be traversed in a maximum of sixty four steps when constructed to a depth of six. The octtree fares reasonably well for such scenes, but cannot really be expected to incur significantly lower navigation costs than the grid partition for decompositions of such shallow depth. Each horizontal step traverses just one voxel, and the number of maximum depth cells contained in each voxel is limited by this depth.

The octtree provides the fastest synthesis for the scenes of highest object count. Such scenes require a higher level of decomposition for adequate local simplification. This cannot be achieved by the memory-intensive grid partition. Moreover, the navigation of such deep decompositions may be expected to be more efficient with the octtree than the grid partition, since far more than three cells of maximum depth may then be traversed in each horizontal step.

In summary, the octtree has a competitive performance for each case study and the best performance for the most complex test pieces. The octtree is predicted to have the best performance for scenes of high object count. Image synthesis times are shown to be *independent* of object count given an octtree of adequate local scene simplification.

Chapter 8: Conclusion

Synopsis:

Chapter eight summarises the research presented in this thesis. Future developments are proposed for the exploitation of various benefits of the octtree in the acceleration of ray tracing and other applications. The chapter ends with the conclusions reached as a result of this thesis.

8.1. Summary of Research Presented in this Thesis

Scene decompositions have been shown to accelerate ray tracing by avoiding an exhaustive object search in the scene model's solution for a given ray [Section 3.4.3]. A ray is considered over successive local regions within such a decomposition rather than simultaneously over the entire scene. The set of objects which are candidates for surface intersection within each such region is greatly reduced from over the global scene. Only the local regions actually navigated by a ray are considered, in path length order. The solution of the scene model need only be considered whilst the current local region is within the significant path length interval [Section 3.2.1].

Three forms of scene decompositions are proposed for the acceleration of ray traced image synthesis. These are the bounding volume hierarchy [Chapter 4], the grid partition [Chapter 5] and the octtree [Chapter 6]. Algorithms are presented for their navigation by a given ray and their automatic generation.

8.2. The Bounding Volume Hierarchy

The bounding volume hierarchy [Chapter 4] has been the subject of much research [Kay,Kajiya;1986: Goldsmith,Salmon;1987; Rubin,Whitted;1980]. The box or other slab-intersection bound is shown to be particularly suited to hierarchy construction since each clipping plane of such a bound is inherited by one of its descendants [Section 4.2].

8.2.1. Navigation of the Bounding Volume Hierarchy

A novel algorithm is presented which exploits inheritance to avoid unnecessary repetition over clipping planes in storage and navigation [Section 4.5]. A binary hierarchy exploits such inheritance best [Section 4.2]. The navigation avoids much of the floating point computation of previous schemes. Many of the bounds missed by a ray are rejected without any floating point multiplication [Section 4.1.2]. Clipping plane intersections with struck bounds are forwarded to descendants to avoid recalculation.

8.2.2. Automatic Generation of the Bounding Volume Hierarchy

An optimal hierarchy should minimise the average number of bounds queried per ray [Section 4.7]. Attention is drawn to the similar behaviour of Huffman's data compression tree which minimises the average number of encryption symbols per encoded datum. Huffman's tree construction is generalised for a bounding volume hierarchy under the assumption that a bound's probability of being navigated by an arbitrary ray is proportional to its surface area [Section 4.8]. A probability inheritance law is established and exploited to avoid an exhaustive search in various tasks.

8.3. The Grid Partition

The grid partition [Chapter 5] has a particularly regular structure which may be navigated with efficient incremental techniques [Section 5.3]. This decomposition has the potential of achieving a greater degree of decomposition than the bounding volume hierarchy [Section 3.5.2].

8.3.1. Navigation of the Grid Partition

Bresenham's incremental line generator for a 2D raster screen is generalised for the navigation of a ray through a grid partition [Section 5.3]. This novel algorithm navigates with three uniformly bounded decision variables in a decision vector. The navigation is maintained incrementally under integer addition and there is no need for exception handling.

8.3.2. Automatic Generation of the Grid Partition

The grid partition is shown to be efficiently generated from an octtree decomposition [Section 5.4]. This exploits scene coherency to avoid the decomposition of scene regions which do not contain any object surfaces. Interval analysis is proposed as a unified framework for the conservative test of a complex CSG object's surface passing through a voxel [Section 5.8]. The grid partition is shown to have potential for lazy construction [Section 5.5].

8.4. The Octtree

The octtree [Chapter 6] is proposed as a suitable decomposition for the acceleration of ray tracing in its own right. Unlike the grid partition, the octtree decomposition adapts to local scene complexity [Section 5.9] and is generally less demanding in memory for a given degree of local scene simplification [Section 6.6].

8.4.1. Navigation of the Octtree

A novel octtree navigation is presented [Section 6.3]. This is based on 'SMARTs' or Spatial Measures for Accelerated Ray Tracing. The octtree is navigated recursively by a series of vertical and horizontal steps in the octtree diagram with two uniformly bounded SMART vectors, an update vector and a comparison variable. The navigation proceeds in efficient integer arithmetic using recurrence relations. Appropriate values are maintained by halving down vertical steps and with increments already in the current environment across horizontal steps. The navigation of horizontal steps is similar to Bresenham's generalisation for the 3D ray navigation of a grid partition [Section 5.3], but abandons the concept of a driving axis and does not take constant width steps.

8.4.2. Automatic Generation of an Octtree

A recursive octtree generation is given with two termination criteria [Section 6.4]. The current voxel is decomposed until the maximum permitted depth is reached or the length of its heterogeneous object list no longer exceeds some simplicity threshold. These twin criteria offer a wider range of decompositions than the simple grid partition. Interval analysis once more provides a conservative test for the surface of a complex CSG object passing through a voxel [Section 5.8]. The octtree is shown to be particularly suited to lazy construction [Section 6.6].

8.5. The Success of the Scene Decompositions

Accelerated ray tracers exploiting all three scene decompositions have been implemented [Section 7]. Numerous images have been synthesised with these, including four case

studies for which experimental results are presented [Section 7.4]. The data are analysed to assess the performance of these scene decompositions [Section 7.5]. Predictions of the influence of object count are also given, and verified with further experimental data [Section 7.7].

8.5.1. The Four Case Studies

Whilst each decomposition dramatically accelerated image synthesis, the Huffman-derived bounding volume hierarchy consistently fared worst. The grid partition provided the fastest image synthesis from the test scenes of low object count, where only a low level of decomposition can provide adequate scene simplification. The octtree had marginally inferior synthesis times for these scenes, but at far lower memory requirement. The octtree came into its own for the test scene of highest object count, where a high level of decomposition was required for adequate simplification. This was infeasible for the memory-intensive grid partition, but proved no problem for the octtree.

8.5.2. Predictions for General Scenes

In the experimentally verified predictions of performance [Section 7.7], construction times for the bounding volume hierarchy are shown to be quadratic in object count. Those for the octtree and grid partition are only linear.

Synthesis times for the bounding volume hierarchy are shown to be logarithmic in object count. Those for the octtree and grid partition are predicted to be independent of object count, given adequate scene simplification. This is verified for the octtree, but proved impractical for the memory-intensive grid partition.

Storage requirement is shown to be linear in object count for the bounding volume hierarchy and octtree. The grid partition's storage requirement is generally independent of object count but greatly exceeds that of the octtree for deep decompositions.

8.5.3. The Success of the Octtree

The octtree appears to offer particularly high performance in the general acceleration of ray traced image synthesis, and provides a flexible degree of scene decomposition. Construction times are linear in object count and memory requirements are generally far less than for the grid partition. Moreover, the octtree is particularly well suited to lazy-construction [Section 6.6]. Image synthesis times are not only greatly reduced by the octtree but may be feasibly made independent of object count [Section 7.7.2].

8.6. Future Developments in the Use of the Octtree

The octtree decomposition has been proposed for the acceleration of ray tracing [Section 6] and has proved particularly successful [Section 7]. However, this is just one example of the many problems to which the octtree is well suited [Samet,Webber;1988: Meagher;1982: Jackins,Tanimoto;1980]. The octtree has great potential for future developments in the acceleration of ray traced image synthesis and other applications.

8.7. Benefits to Image Synthesis

8.7.1. Further Methods of Accelerating Ray Tracing

Image synthesis times may be made independent of object count by the octtree, but are still influenced by the complexity of CSG objects. However the octtree decomposition may be developed to simplify objects as well as scenes.

A bounding volume hierarchy may decompose a CSG object into component constructs of local significance [Section 3.2.3]. In the current octtree implementation, the entire specification of any heterogeneous object is significant within a voxel. However the object may be decomposed within the voxel by CSG tree pruning [Wyvill et al;1986: Woodwark,Bowyer;1986]. This simplifies the consideration of the object in any further decomposition of the voxel or query by a ray navigating the voxel. An octtree decomposition of a CSG dice model is shown [Fig 8.7.1a]. The body is modelled as the intersection of a sphere and cube, and the holes by the subtraction of several smaller

Fig 8.7.1a: Non-Empty Voxels in an Octtree Decomposition of a CSG Dice Model

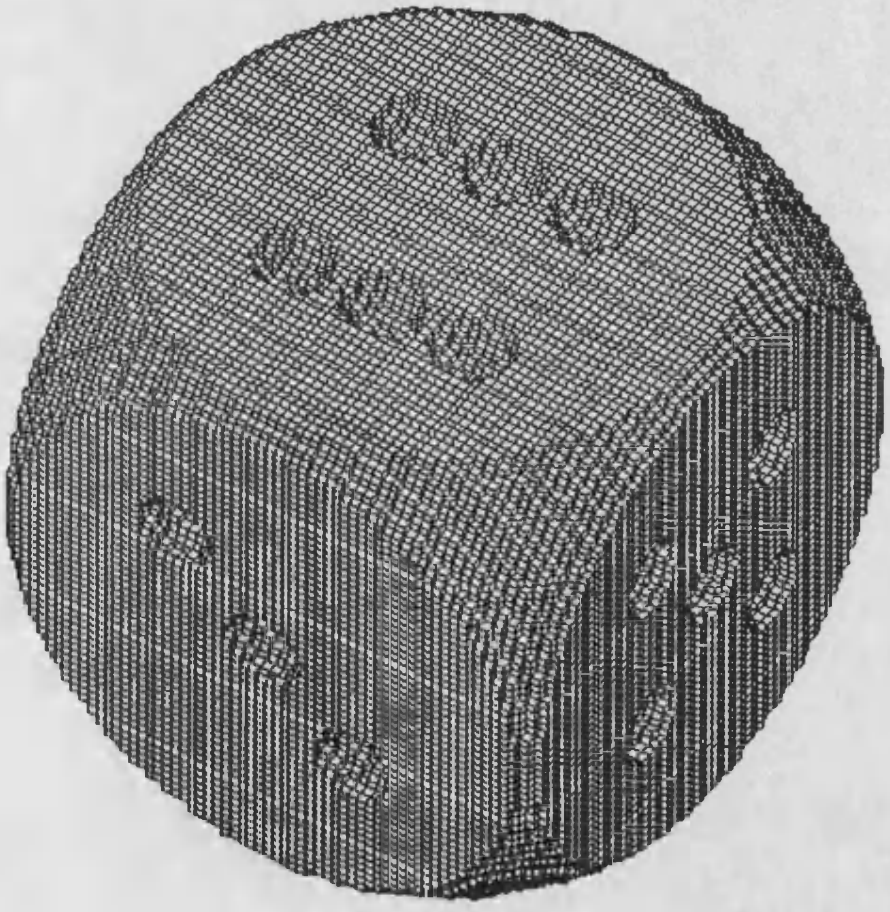


Fig 8.7.1a

spheres. The complete object is modelled with twenty three primitives. However, in a majority of leaf voxels only one primitive is heterogeneous and never more than two. The complex CSG tree description of the dice may be pruned within each voxel to a simpler form involving only the heterogeneous primitives. An object's state code is classified within a voxel by recursion over the object's CSG tree [Section 5.8.1]. Interval analysis is applied at each node. If the height interval does not contain zero, then the construct modelled by this node is known to be homogeneous with respect to the voxel and may be pruned. Some care must be taken when pruning a branch of a subtraction or symmetric difference node. If the branch is homogeneously inside the voxel, its sibling must be complemented to ensure the construct maintains the same sense. The local CSG description of an object within a voxel may be greatly simplified in this manner. However the pruned description is only relevant within the local voxel. Any ray navigating this voxel must therefore ignore any intersection beyond the voxel. The previous method for avoiding repeated queries of global object specifications [Section 5.1.1] is no longer applicable, and so the same object may be queried several times by the same ray [Wyvill et al;1986].

8.7.2. The Unification of Lazy Construction with Image Synthesis

Octtrees are particularly suitable for lazy construction during image synthesis [Section 6.6]. Only regions navigated by a ray need actually be decomposed, thus avoiding the time and storage requirement of unnecessary decomposition. A greater degree of decomposition can be achieved within given time and storage space. Voxels may be progressively decomposed to ever greater depths as navigated by rays. Eventually, the sizes of the current voxel and the height interval estimates for the associated heterogeneous objects will become negligible within some given precision. The voxel centre is then known to be in close proximity of a height function root, due to the precision of the height interval. Any ray navigating such a small voxel is known to come close to the voxel centre. The ray is therefore known to come into close approach of a height function root, being within the given precision in both the voxel domain and height interval codomain. Any attempt at the numerical isolation of a root within the voxel to that given precision would immediately

succeed without a single iteration, due to this proximity. Lazy construction may therefore be used as a numerical method for root isolation to given precision. Roots are directly isolated in 3D, as opposed to other numerical methods restricted to the single dimension of a given ray's path [Section 3.2.4]. This root location does not require solution by radicals and becomes unified with scene decomposition to 'kill two birds with one stone'. Such extended decomposition would effectively convert the CSG model to a direct octtree encoding during image synthesis.

Root isolation may be expected to be less efficient in 3D than 1D for any given ray. However, many coherent rays are traced during image synthesis. Whilst 1D root isolation considers different rays independently, the 3D method shares much computation between coherent rays. Such rays pass through similar voxels, and the decomposition generated for the first need not be repeated for any of the others. Root isolation by extended lazy decomposition would therefore exploit ray as well as object coherency.

8.7.3. The Application of Octtrees in Generalised Scene Models

Some CAD/CAM researchers have introduced extra primitives as 'blends' in CSG solid models [Zhang,Bowyer;1986]. These blends are automatically generated between the constructs in a CSG object. Blends with height functions of up to degree sixteen have been used. The roots of such height functions may be found numerically with Sturm's method [Section 3.2.4] for independent rays. However, the interval analysis of 3D root isolation by lazy decomposition is of general application and could also exploit ray coherency in the isolation of roots to these higher degree polynomials.

The CSG model is suitable for the design of objects from a given set of primitives. However, some applications require the synthesis of an image from an existing object which is not easily specified with this model. In medical applications, bone structure data may be obtained from the body scan of a patient with complicated fractures [Frenkel;1989]. The surgeon may wish to visualise the extent of injury before operating. Such data may be directly modelled with an octtree encoding from which an image may be synthesised. The octtree may therefore model scenes directly [Samet,Webber;1988: Meagher;1982:

Jackins,Tanimoto;1980] as well as locally simplifying the CSG scene model. The intersection of a ray with an object modelled in this manner is easily found. The ray is simply navigated through the octtree until reaching a voxel within the object's specification.

8.8. Benefits to CAD/CAM

The synthesis of an image is but one of the many problems in CAD/CAM to be solved from an object model. The object's volume may be required for casting from a mould. An object's mass may also be needed and can be found from its volume and density. The centre of gravity or moment of inertia may also have to be checked before manufacture. The octtree model provides particularly simple solutions to such problems even for complex objects.

A CSG object may be approximated by an octtree decomposition constructed by interval analysis. A deep decomposition under a simplicity criterion of zero will produce a close approximation. Whereas a CSG object may be specified in terms of various primitives, only the simple voxel is used in the octtree approximation. Objects may also be modelled directly by an octtree encoding. Many problems are easily solved for the octtree model with finite element analysis which need only address the simple voxel.

A lower bound on an object's volume is easily calculated as the summed volume of all leaf octtree voxels homogeneously inside the object. A maximum error term is calculated as the summed volume of all leaf voxels over which the object is heterogeneous. This error term decreases with octtree refinement to disappear in the limit. An estimate of volume for a given decomposition may be taken as the lower bound plus half the maximum error term.

The object's mass may be estimated by multiplying this volume by the object's density. Centre of gravity may be estimated as the vector sum of the centres of leaf voxels homogeneously inside the object, weighted by their volume. Moment of inertia may be estimated similarly.

8.9. Benefits to Animation

Octtree decompositions provide solutions to various problems in animated image synthesis.

8.9.1. Collision Detection at the Frame Level

Collision detection is a difficult problem when addressing a complex CSG scene model directly. Whilst possible collisions are easily specified as object intersections, determining whether such intersections are non-empty remains complicated. Once more, an approximation by octtree decomposition provides a simple solution by addressing only the simple voxel rather than a range of primitives.

Any voxel in a scene's octtree decomposition which is found to be homogeneously inside both of two objects is known to be within their intersection, and thereby indicates a collision. Any voxel which is not homogeneously outside either of two objects may possibly be part of their intersection and indicates a possible collision. Further decomposition of the voxel may prove for certain whether a collision actually occurs.

8.9.2. Hex-tree Decompositions for Collision Detection over Time

The previous scheme detects collisions at given 'snap-shot' frames but not between these frames. The duration of a collision between two objects travelling quickly in opposite directions may not include such a frame. The objects in an animated synthesis of the scene will appear to pass through each other without a collision being detected at any frame. The temporal sub-sampling of such a scene is clearly subject to aliasing.

Temporal aliasing may be overcome by decomposing an animated scene not only in the three spatial dimensions but also in time as a fourth dimension [Glassner;1988]. This results in a *hex-tree* since each decomposed voxel is bisected in four dimensions to spawn sixteen children. The animated scene is divided into *time-voxels* of specified temporal as well as spatial extent. Any time-voxel found to be homogeneously inside both of two objects indicates that the spatial voxel is contained within their intersection over the associated time interval. A definite collision is then detected over this time interval. Any time-voxel not homogeneously outside either of two dynamic objects indicates a possible

such collision.

Glassner [1988] has implemented a hex-tree decomposition scheme, but resorted to a hybrid of octtree decomposition and bounding volume techniques. This scheme is inappropriate for collision detection since time-space bounds are prone to over-approximation, particularly for fast moving objects. Such bounds may have to allow for dynamic changes in size and shear as well as orientation and position. Glassner's scheme was limited to a means of computational savings in the synthesis of dynamic images, including motion blur.

Interval analysis [Section 5.8] could be generalised to provide a hex-tree construction less prone to such over-approximation and well suited to dynamic collision detection. The dynamic change of a CSG object may be modelled by assigning dynamic instancing transformations to the primitives from which it is constructed. The deformation matrix and shift vector associated with a primitive instance are then no longer constant, but vary with time. The shift vector may typically follow a spline path, each component being a cubic in time. The elements of the deformation matrix may be specified by similar cubics and/or trigonometric functions in time. These transformations may then be considered over *time intervals* rather than being constant. The linear algebra of world to local transformations may be generalised to allow for dynamic transformations. The spatial voxel bound of a time-voxel's local image may be found over the associated time interval. All dynamic changes may be modelled by this transformation. The remaining interval analysis for the hex-tree decomposition need only address spatial intervals as before [Section 5.8].

8.9.3. Hex-trees for Reduced Animation Costs

Hex-trees need only be constructed *once* for an entire animated scene rather than repeatedly for each frame. Construction costs may be expected to be significantly reduced in the same manner as Glassner's scheme [Glassner;1988]. Lazy decomposition may be exploited as before [Section 8.7.2]

8.9.4. Hex-trees for Motion Blur

Ray tracing is a sampling technique and is therefore prone to *aliasing* [Amanatides;1987]. Anti-aliasing is often achieved by super-sampling. Octtrees are suitable for spatial super-sampling. Several rays may be efficiently traced through different points of a pixel which is then assigned the average visible colour. Hex-trees are suitable for not only spatial but also temporal super-sampling, allowing the synthesis of motion blur. Several rays may be efficiently traced through a pixel at different times within some exposure interval. The pixel is assigned the average colour once more. The same hex-tree may be navigated for rays at any time within the decomposed continuum. Assuming objects move at speeds which are insignificant compared to that of light, each ray need only traverse the spatial dimensions within the hex-tree. Hex-trees may therefore be navigated with the SMART algorithm described for the octtree [Section 6.3].

8.10. Benefits to Parallel Processing

Various researchers have ray traced synthetic images on parallel processing systems [Section 3.4]. Ray tracing synthesises realism by considering each ray independently, and is therefore highly parallelisable.

The synthesis of an image may be divided amongst processors in various ways. The 2D image may be divided into several sub-images, and each synthesised independently by a different processor. However, this requires each processor to hold the entire scene model, which may be large for complex scenes. This is not always feasible if each processor has only a limited amount of memory. The 3D scene may be divided instead into the voxel regions of an octtree decomposition, each of which is dealt with by an independent processor. This would reduce the amount of memory required for the local scene description at each processor. Rays could be navigated between these voxels by other processors. This scheme needs development to address problems of processor communication, load balancing and so on but seems worthy of such research.

8.11. Conclusions from this Thesis

The major conclusions to be drawn from this thesis address the suitability of the various scene decompositions for accelerating ray traced image synthesis.

Bounding volume hierarchies have been the subject of much recent research [Section 4]. These do indeed offer substantial savings over naive ray tracing, but generally prove inferior to scene partitions such as the regular grid and octtree decomposition.

The grid partition can greatly accelerate image synthesis [Section 5]. Efficient navigation and construction algorithms have been presented in this thesis. However, the grid partition proves too demanding in memory for the adequate simplification of particularly complex scenes.

The octtree decomposition performs particularly well in the acceleration of image synthesis, being efficient in construction, navigation and storage requirement [Section 6]. Many potential benefits may be developed in the future. The octtree seems worthy of further research in the acceleration of ray traced image synthesis and other applications.

APPENDIX A: Linear Algebra for Transforms between Local and World Space

Synopsis:

Each primitive instance in world space has an associated linear transform from local space. The user provides this transform for position vectors. Appendix A derives various other vector transforms between local and world space required for ray tracing. These are the inverse position transform from world to local space, the direction transform from world to local space and the surface normal transform from local to world space.

Derivation

Any linear transform can be split into two components. On one hand, a concentric deformation component gives the net effect of rotations, scalings, reflections and shears all about the origin. This is modelled by a 3×3 matrix with respect to Cartesian bases. On the other, a translation component allows for a net shift in origin. This is modelled with the appropriate 3D shift vector. These components must be applied in the correct order to produce the transform as their product is not necessarily commutative. Each primitive instance in the scene model is attributed an appropriate deformation matrix and translation vector to world space. Related transformations are derived as necessary.

Notation

Symbol	Meaning
$W = [\underline{w}_1, \underline{w}_2, \underline{w}_3]$	Local to world deformation matrix for points, described as column vectors
$\underline{c}$	Local to world translation vector for points
L	World to local deformation matrix for points
D	World to local deformation matrix for direction vectors
S	Local to world deformation matrix for surface normals
$\underline{p}_w$	World point
$\underline{p}_l$	Local point
$\underline{d}_w$	World direction vector
$\underline{d}_l$	Local direction vector
$\underline{n}_w$	World surface normal vector
$\underline{n}_l$	Local surface normal vector

Local to World Point Transform

Each primitive instance is attributed transform components such that

$$\underline{p}_w = \underline{c} + W\underline{p}_l = \underline{c} + p_{lx}\underline{w}_1 + p_{ly}\underline{w}_2 + p_{lz}\underline{w}_3$$

by definition of matrix multiplication.

Local to World Surface Normal Transform

Surface normal vectors are subject only to a deformation during transformation, not translation. They do not necessarily undergo the same deformation as position vectors. Consider a cube sheared sideways about its centre. Whilst the vertical position vectors from the centre to the upper and lower faces are sheared over, the vertical surface normal vectors along these faces are not. However, surface normals undergo a deformation related to that for position vectors.

Consider the local surface normal $\underline{n_l}$ at local surface position $\underline{p_l}$. Let $\underline{a}$, $\underline{b}$ be two linearly independent direction vectors in the surface tangent plane at $\underline{p_l}$ with $\underline{a}\times\underline{b}$ in the same direction as $\underline{n_l}$. The local tangent plane

$$\underline{p_l} + \sum_{\alpha,\beta\in R} \alpha\underline{a}+\beta\underline{b}$$

transforms position-wise to a world tangent plane

$$\underline{c} + W\left[\underline{p_l} + \sum_{\alpha,\beta\in R} \alpha\underline{a}+\beta\underline{b}\right] = \underline{p_w} + \sum_{\alpha,\beta\in R} \alpha W\underline{a}+\beta W\underline{b}$$

This world tangent plane at $\underline{p_w}$ has normal $\underline{n_w}$ such that

$\underline{n_w}$ is in the same direction as

$W\underline{a} \times W\underline{b}$

$= (a_x\underline{w_1} + a_y\underline{w_2} + a_z\underline{w_3}) \times (b_x\underline{w_1} + b_y\underline{w_2} + b_z\underline{w_3})$

$= (a_yb_z-a_zb_y)(\underline{w_2}\times\underline{w_3}) + (a_zb_x-a_xb_z)(\underline{w_3}\times\underline{w_1}) + (a_xb_y-a_yb_x)(\underline{w_1}\times\underline{w_2})$

$= (\underline{a}\times\underline{b})_x\underline{s_1} + (\underline{a}\times\underline{b})_y\underline{s_2} + (\underline{a}\times\underline{b})_z\underline{s_3}$ where $\underline{s_1} = \underline{w_2} \times \underline{w_3}$; $\underline{s_2} = \underline{w_3} \times \underline{w_1}$; $\underline{s_3} = \underline{w_1} \times \underline{w_2}$

is in the same direction as

$n_{l_x}\underline{s_1} + n_{l_y}\underline{s_2} + n_{l_z}\underline{s_3}$

$\rightarrow$ $\underline{n_w}$ is in the same direction as

$S\underline{n_l}$

where $S = [\underline{s_1},\underline{s_2},\underline{s_3}]$ is the matrix of cofactors of W, cofactor(W).

A unit world surface normal vector is found by an appropriate scaling of $\underline{n_w}$.

World to Local Point Transform

From the primitive instance,

$$\underline{p_w} = \underline{c} + W\underline{p_l}$$

$$\rightarrow \underline{p_w} - \underline{c} = W\underline{p_l}$$

$$\rightarrow \underline{p_l} = L(\underline{p_w} - \underline{c})$$

where $L = W^{-1}$ and by standard matrix theory,

$$W^{-1} = \frac{1}{\det(W)}\text{cofactor}^T(W) = \frac{1}{\det(W)}S^T$$

where $\det(W) = \underline{w_1} \cdot [\underline{w_2} \times \underline{w_3}]$, the scalar triple product of its column vectors.

World to Local Direction Transform

Direction vectors are subject only to a deformation during transformation, not translation. They undergo the same deformation as position vectors. Therefore

$$\underline{d_w} = W\underline{d_l}$$

$$\rightarrow \underline{d_l} = D(d_w)$$

where $D = W^{-1} = L$

Transform Summary

$$\text{LOCAL} \left\{ \begin{array}{lcl} \rightarrow & p_w = \underline{c} + W\underline{p_l} & \rightarrow \\ \rightarrow & n_w = S\underline{n_l} & \rightarrow \\ \leftarrow & p_l = L(\underline{p_w} - \underline{c}) & \leftarrow \\ \leftarrow & d_l = D\underline{d_w} & \leftarrow \end{array} \right\} \text{WORLD}$$

where

W, $\underline{c}$ are defined by primitive instance

$S = \text{cofactor}(W)$

$$L = W^{-1} = \frac{1}{\det(W)}S^T$$

$D = L$

APPENDIX B: Height Functions

Synopsis:

Appendix B derives height functions defined over local space for a range of primitives. These are generalised to a definition over world space for boolean constructs

Derivation

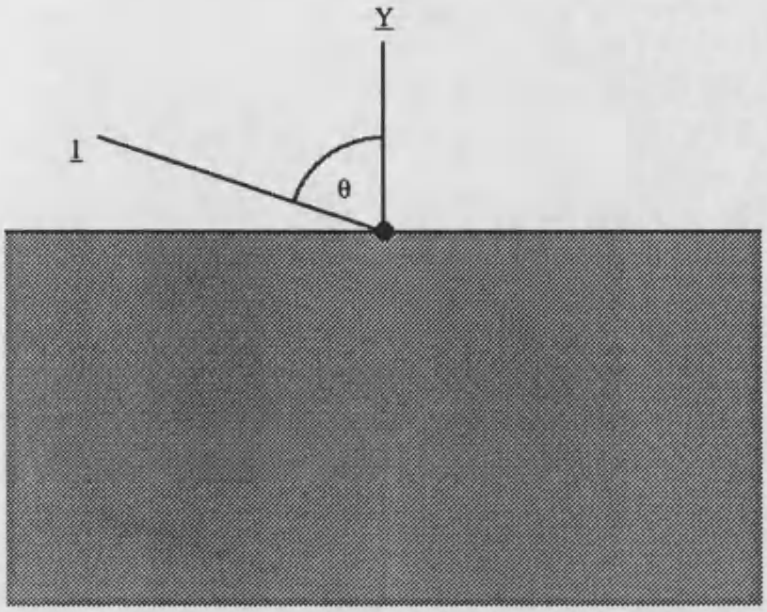
Each local primitive is specified as the intersection of simple geometries. Each geometry is described by a characteristic polynomial in local coordinates specifying a notion of height above surface. The polynomial is negative inside the geometry, positive outside and zero on the surface. Each local primitive's height function is taken as the maximum of those over the intersected geometries.

Local height functions are derived for the plane, cube, sphere, cylinder, cone and torus. By convention each local primitive is centred at the local origin and is aligned with the local axes. The coordinate system is in a left-handed sense with increasing X going from left to right, Y from down to up and Z from behind to infront. All diagrams are shown in a local plane containing the origin and Y axis. These height functions are generalised to a definition over boolean union and complement and thereby to a range of boolean operations.

Notation

Symbol	Meaning
$\underline{l}$	Local point
$\theta \in [0,\pi]$	Angle between local point and primitive axis.
A, B	Boolean constructs
X'	Complement of boolean construct X
Height_X	World height above boolean construct X
$\underline{w}$	World point

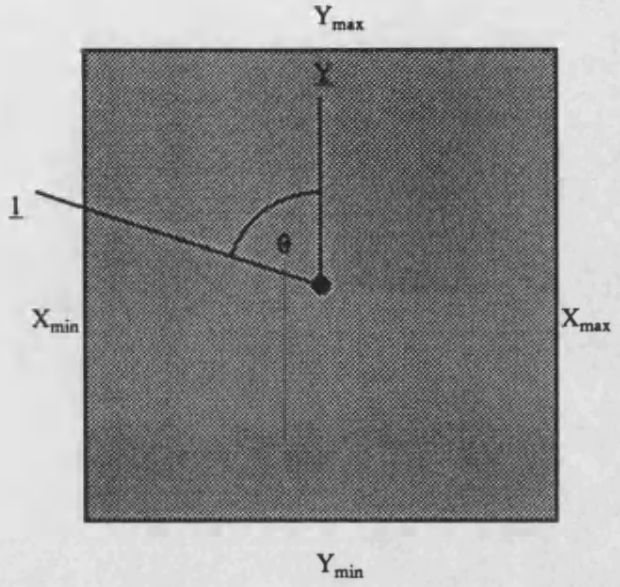
The Plane



$$\text{LOCAL\_PLANE} = \left\{ \underline{l} : l_y < 0 \right\}$$

$$\text{Local\_Plane}(\underline{l}) = l_y$$

The Cube of X extent $[X_{min},X_{max}]$, Y extent $[Y_{min},Y_{max}]$, Z extent $[Z_{min},Z_{max}]$

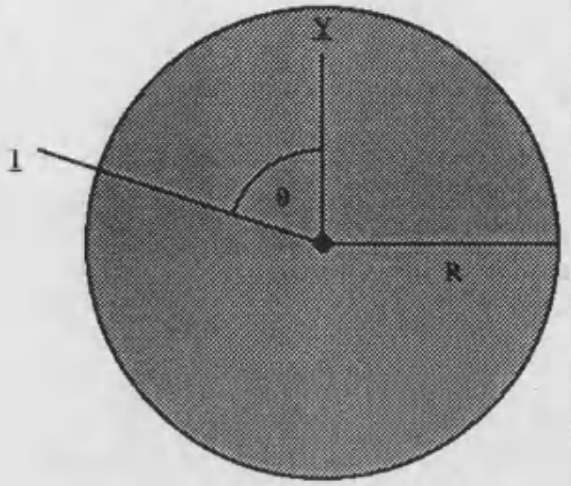


$$\text{LOCAL\_CUBE} = \left\{ \underline{l} : l_x \in [X_{min},X_{max}],\ l_y \in [Y_{min},Y_{max}],\ l_z \in [Z_{min},Z_{max}] \right\}$$

$$= \left\{ \underline{l} : \begin{array}{c} l_x - X_{max} < 0,\ X_{min} - l_x < 0 \\ l_y - Y_{max} < 0,\ Y_{min} - l_y < 0 \\ l_z - Z_{max} < 0,\ Z_{min} - l_z < 0 \end{array} \right\}$$

$$\text{Local\_Cube}(\underline{l}) = \max \left\{ \begin{array}{c} \max\ \{l_x - X_{max},\ X_{min} - l_x\}, \\ \max\ \{l_y - Y_{max},\ Y_{min} - l_y\}, \\ \max\ \{l_z - Z_{max},\ Z_{min} - l_z\} \end{array} \right\}$$

The Sphere of Radius R

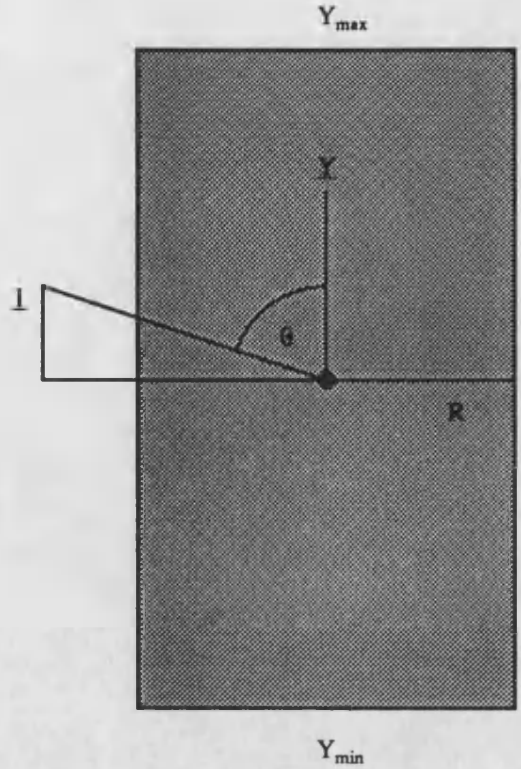


$$\text{LOCAL\_SPHERE} = \left\{ \underline{l} : |\underline{l}| < R \right\} = \left\{ \underline{l} : \underline{l}.\underline{l} - R^2 < 0 \right\}$$

$$\text{Local\_Sphere}_R(\underline{l}) = \underline{l}.\underline{l} - R^2$$

$$= l_x^2 + l_y^2 + l_z^2 - R^2$$

The Cylinder of Radius R, Y extent $[Y_{min}, Y_{max}]$



$$\text{LOCAL\_CYLINDER} = \left\{ \underline{l} : |\underline{l}|\sin(\theta) < R,\ l_y \in [Y_{min}, Y_{max}] \right\}$$

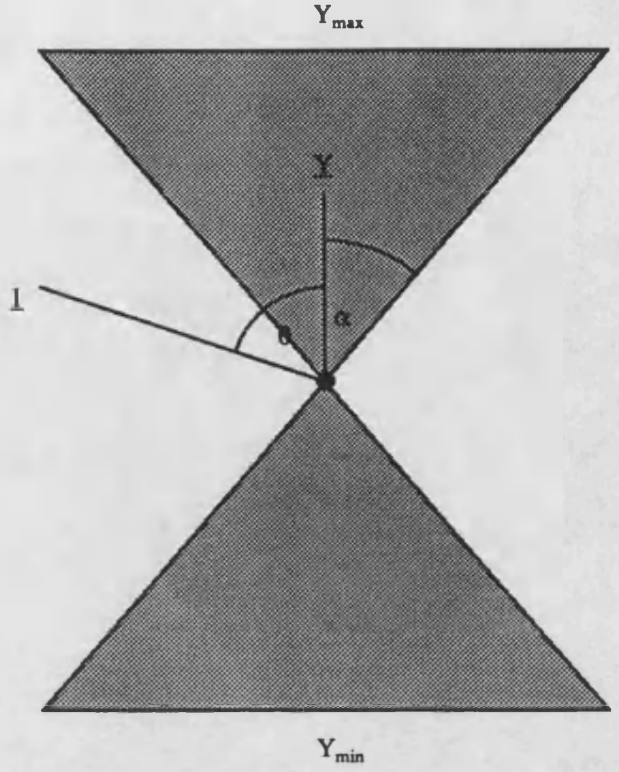
$$= \left\{ \underline{l} : (\underline{l}.\underline{l})\sin^2(\theta) - R^2 < 0,\ l_y - Y_{max} < 0,\ Y_{min} - l_y < 0 \right\}$$

$$= \left\{ \underline{l} : \underline{l}.\underline{l} - l_y^2 - R^2 < 0,\ l_y - Y_{max} < 0,\ Y_{min} - l_y < 0 \right\}$$

$$\text{Local\_Cylinder}_R(\underline{l}) = \max \left\{ \begin{matrix} \underline{l}.\underline{l} - l_y^2 - R^2, \\ \max\ \{l_y - Y_{max},\ Y_{min} - l_y\} \end{matrix} \right\}$$

$$= \max \left\{ \begin{matrix} l_x^2 + l_z^2 - R^2, \\ \max\ \{l_y - Y_{max},\ Y_{min} - l_y\} \end{matrix} \right\}$$

The Double Cone of Axial Angle α, Y extent $[Y_{min},Y_{max}]$



$$\text{LOCAL\_CONE} = \left\{ \underline{l} : \theta<\alpha \text{ or } \theta>\pi-\alpha,\ l_y \in [Y_{min},Y_{max}] \right\}$$

$$= \left\{ \underline{l} : |\underline{l}|^2\cos^2(\theta)>|\underline{l}|^2\cos^2(\alpha),\ l_y \in [Y_{min},Y_{max}] \right\}$$

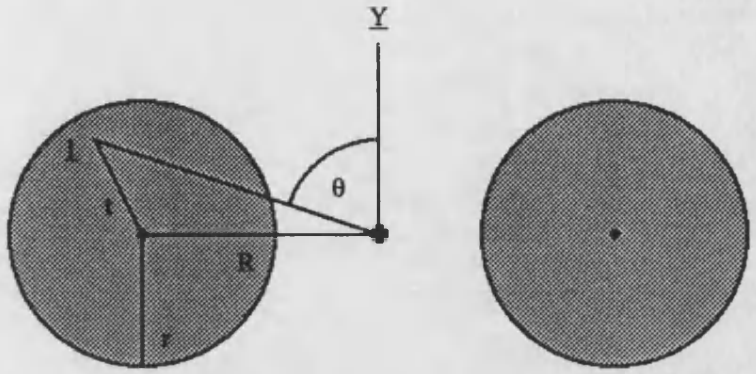
$$= \left\{ \underline{l} : l_y^2>\underline{l}.\underline{l}\cos^2(\alpha),\ l_y \in [Y_{min},Y_{max}] \right\}$$

$$= \left\{ \underline{l} : \underline{l}.\underline{l}\cos^2(\alpha) - l_y^2 < 0,\ l_y - Y_{max} < 0,\ Y_{min} - l_y < 0 \right\}$$

$$\text{Local\_Cone}_\alpha(\underline{l}) = \max \left\{ \begin{matrix} (\underline{l}.\underline{l})\cos^2(\alpha)-l_y^2, \\ \max\ \{\ l_y - Y_{max},\ Y_{min} - l_y\} \end{matrix} \right\}$$

$$= \max \left\{ \begin{matrix} (l_x^2+l_z^2)\cos^2(\alpha)-l_y^2\sin^2(\alpha), \\ \max\ \{\ l_y - Y_{max},\ Y_{min} - l_y\} \end{matrix} \right\}$$

The Torus of Major Axis R, Minor Axis r



By the cosine rule

$$t^2 = R^2 + |\underline{l}|^2 - 2R|\underline{l}|\cos(\frac{\pi}{2}-\theta) = R^2 + \underline{l}.\underline{l} - 2R|\underline{l}|\sin(\theta)$$

$$\text{LOCAL\_TORUS} = \left\{ \underline{l} : t < r \right\} = \left\{ \underline{l} : t^2 < r^2 \right\} = \left\{ \underline{l} : R^2 + \underline{l}.\underline{l} - 2R|\underline{l}|\sin(\theta) < r^2 \right\}$$

$$= \left\{ \underline{l} : \left[(R^2-r^2) + \underline{l}.\underline{l} \right]^2 < 4R^2|\underline{l}|^2\sin^2(\theta) \right\}$$

$$= \left\{ \underline{l} : (R^2-r^2)^2 + 2(R^2-r^2)\underline{l}.\underline{l} + \underline{l}.\underline{l}^2 < 4R^2\underline{l}.\underline{l} - 4R^2 l_y^2 \right\}$$

$$= \left\{ \underline{l} : (R^2-r^2)^2 - 2(R^2+r^2)\underline{l}.\underline{l} + \underline{l}.\underline{l}^2 + 4R^2 l_y^2 < 0 \right\}$$

$$= \left\{ \underline{l} : \left[\underline{l}.\underline{l} - (R^2+r^2) \right]^2 - (R^2+r^2)^2 + (R^2-r^2)^2 + 4R^2 l_y^2 < 0 \right\}$$

$$\text{Local\_Torus}_{R,r}(\underline{l}) = \left[\underline{l}.\underline{l} - (R^2+r^2) \right]^2 + 4R^2(l_y^2 - r^2)$$

Boolean Operations

Union:

$$\underline{w} \in A \text{ union } B \leftrightarrow \underline{w} \in A \text{ or } \underline{w} \in B$$

$$\text{so } Height_{A \text{ union } B}(\underline{w}) < 0 \leftrightarrow Height_A(\underline{w}) < 0 \text{ or } Height_B(\underline{w}) < 0$$

$$\text{Therefore } Height_{A \text{ union } B} = \min(\, Height_A, Height_B)$$

Complementation:

$$\underline{w} \in A' \leftrightarrow \underline{w} \notin A$$

$$\text{so } Height_{A'}(\underline{w}) < 0 \leftrightarrow Height_A(\underline{w}) \geq 0$$

$$\text{Therefore } Height_{A'} = -Height_A$$

Intersection: By DeMorgan's Rules,

$$A \text{ intersect } B = (\, A' \text{ union } B' \,)'$$

$$\text{Therefore } Height_{A \text{ intersect } B} = Height_{(\, A' \text{ union } B' \,)'} = -Height_{A' \text{ union } B'}$$

$$= -\min(Height_{A'}, Height_{B'}) = -\min(-Height_A, -Height_B) = \max(Height_A, Height_B)$$

Subtraction:

$$A \text{ subtract } B = A \text{ intersect } B'$$

$$\text{Therefore } Height_{A \text{ subtract } B} = Height_{A \text{ intesect } B'}$$

$$= \max(Height_A, Height_{B'}) = \max(Height_A, -Height_B)$$

Symmetric Difference:

$$A \text{ difference } B = (\, A \text{ union } B \,) \text{ subtract } (\, A \text{ intersect } B \,)$$

$$\text{Therefore } Height_{A \text{ difference } B} = Height_{(\, A \text{ union } B \,) \text{ subtract } (\, A \text{ intersect } B \,)}$$

$$= \max(Height_{A \text{ union } B}, -Height_{A \text{ intersect } B}) = \max(\min(Height_A, Height_B), -\max(Height_A, Height_B))$$

APPENDIX C: Algebra for Ray Intersection

Synopsis:

Appendix C derives univariate polynomials in ray path length for height above a range of primitive surfaces in local space. These are generalised to univariate height functions in ray path length for boolean constructs. The intersection of a world space ray with a primitive instance is found by transforming to its local space [APPENDIX A]. The ray is outside the primitive when the height is positive, inside when negative, and intersects the surface at the appropriate polynomial roots. These roots may be found analytically for polynomials up to degree four or with iterative numerical techniques for any order.

Derivation

Each local primitive has an associated height function defined in local space [APPENDIX B]. This is the maximum taken over a number of polynomials for the simple geometries intersecting to form the primitive. The height above the primitive along a local space ray is found by substituting the local ray equation into these polynomials, rearranging to a univariate polynomial in the ray's path length, and taking the maximum. A linear polynomial results for the plane, three pairs of linear polynomials for the cube, a quadratic for the sphere, a quadratic and a pair of linear polynomials for the cylinder and cone and a quartic for the torus. Each quadratic has a constant factor of two in the linear term which divides out all constant terms from the root equation. Some terms in the polynomial's coefficients are constant so need only be calculated once per object rather than repeatedly for each ray. If a height function proves to have no real positive root the surface in question is not intersected by the ray.

The intersection of a ray with a boolean construct is found by selecting the appropriate roots from the intersections with the combined arguments.

Notation

Symbol	Meaning
$\underline{l}$	Local point
$\underline{s}$	Local ray source
$\underline{d}$	Local ray direction
A, B	Boolean constructs
$Height_X$	World height above boolean construct X
R_X	Set of path length roots for surface intersections with construct X

The ray is modelled in local space by Local_Ray = $\{ \underline{s}+\lambda\underline{d} : \lambda>0 \}$. The local image of a world ray is found with the appropriate linear algebra [APPENDIX A]. Whilst the view model generates each world ray with a unit direction vector, the direction vector of the local image is not necessarily of unit length.

Each path length root to a local primitive height function corresponds to a local surface intersection. This local point may be found by substituting the root into the local ray equation. The corresponding world surface point could be found by transforming the local point to world space. However the linearity of transformations between local and world space allows the world surface point to be found by direct substitution of the path length root into the world ray equation.

The Plane

$$\text{Local\_Plane}(\underline{l}) = l_y$$

$$\text{Local\_Plane}(\underline{s}+\lambda\underline{d}) = (\underline{s}+\lambda\underline{d})_y$$

$$= \lambda\,[d_y] + [\underline{s}_y]$$

linear form : $\lambda A + B$

$$\text{root} : \lambda = -\frac{B}{A}$$

The Cube of X extent $[X_{min},X_{max}]$, Y extent $[Y_{min},Y_{max}]$, Z extent $[Z_{min},Z_{max}]$

$$\text{Local\_Cube}(\underline{l}) = \max \begin{Bmatrix} \max\{l_x - X_{max},\ X_{min} - l_x\}, \\ \max\{l_y - Y_{max},\ Y_{min} - l_y\}, \\ \max\{l_z - Z_{max},\ Z_{min} - l_z\} \end{Bmatrix}$$

$$\text{Local\_Cube}(\underline{s}+\lambda\underline{d}) = \max \begin{Bmatrix} \max\{(\underline{s}+\lambda\underline{d})_x - X_{max}, X_{min} - (\underline{s}+\lambda\underline{d})_x\}, \\ \max\{(\underline{s}+\lambda\underline{d})_y - Y_{max}, Y_{min} - (\underline{s}+\lambda\underline{d})_y\}, \\ \max\{(\underline{s}+\lambda\underline{d})_z - Z_{max}, Z_{min} - (\underline{s}+\lambda\underline{d})_z\} \end{Bmatrix}$$

$$= \max \begin{Bmatrix} \max\{\lambda[d_x] + [s_x - X_{max}],\ \lambda[-d_x] + [X_{min} - s_x], \\ \max\{\lambda[d_y] + [s_y - Y_{max}],\ \lambda[-d_y] + [Y_{min} - s_y], \\ \max\{\lambda[d_z] + [s_z - Z_{max}],\ \lambda[-d_z] + [Z_{min} - s_z] \end{Bmatrix}$$

linear forms : $\lambda A + B$

$$\text{root} : \lambda = -\frac{B}{A}$$

The Sphere of Radius R

$$\text{Local\_Sphere}_R(\underline{l}) = \underline{l}.\underline{l} - R^2$$

$$\text{Local\_Sphere}_R(\underline{s}+\lambda\underline{d}) = (\underline{s} + \lambda\underline{d}).(\underline{s} + \lambda\underline{d}) - R^2$$

$$= \lambda^2\underline{d}.\underline{d} + \lambda\, 2\,[\underline{s}.\underline{d}] + [\underline{s}.\underline{s} - R^2]\ (R^2 \text{ constant})$$

quadratic form : $\lambda^2 A + \lambda 2B + C$

$$\text{roots} : \lambda = \frac{-B \pm \sqrt{B^2 - AC}}{A}\ (\text{linear form when } A = 0)$$

The Cylinder of Radius R, Y extent $[Y_{min},Y_{max}]$

$$\text{Local\_Cylinder}_R(\underline{l}) = \max \begin{Bmatrix} \underline{l}.\underline{l} - l_y^2 - R^2, \\ \max\{l_y - Y_{max},\ Y_{min} - l_y\} \end{Bmatrix}$$

$$\text{Local\_Cylinder}_R(\underline{s}+\lambda\underline{d}) = \max \begin{Bmatrix} (\underline{s}+\lambda\underline{d}).(\underline{s}+\lambda\underline{d}) - (\underline{s}+\lambda\underline{d})_y^2 - R^2 \\ \max\{(\underline{s}+\lambda\underline{d})_y - Y_{max},\ Y_{min} - (\underline{s}+\lambda\underline{d})_y\} \end{Bmatrix}$$

$$= \max \begin{Bmatrix} \lambda^2[\underline{d}.\underline{d} - d_y^2] + \lambda 2[\underline{s}.\underline{d} - s_y d_y] + [\underline{s}.\underline{s} - s_y^2 - R^2] \\ \max\{\lambda[d_y] + [s_y - Y_{max}],\ \lambda[-d_y] + [Y_{min} - s_y]\} \end{Bmatrix}$$

quadratic form : $\lambda^2 A + \lambda 2B + C$

roots : $\lambda = \dfrac{-B \pm \sqrt{B^2 - AC}}{A}$ (linear form when $A = 0$)

linear forms : $\lambda A + B$

root : $\lambda = -\dfrac{B}{A}$

The Double Cone of Axial Angle α, Y extent $[Y_{min},Y_{max}]$

$$\text{Local\_Cone}_R(\underline{l}) = \max \begin{Bmatrix} \underline{l}.\underline{l}\cos^2(\alpha) - l_y^2, \\ \max\{l_y - Y_{max},\ Y_{min} - l_y\} \end{Bmatrix}$$

$$\text{Local\_Cone}_R(\underline{s}+\lambda\underline{d}) = \max \begin{Bmatrix} (\underline{s}+\lambda\underline{d}).(\underline{s}+\lambda\underline{d})\cos^2(\alpha) - (\underline{s}+\lambda\underline{d})_y^2, \\ \max\{(\underline{s}+\lambda\underline{d})_y - Y_{max},\ Y_{min} - (\underline{s}+\lambda\underline{d})_y\} \end{Bmatrix}$$

$$= \max \begin{Bmatrix} \lambda^2[\underline{d}.\underline{d}\cos^2(\alpha) - d_y^2] + \lambda 2[\underline{s}.\underline{d}\cos^2(\alpha) - s_y d_y] + [\underline{s}.\underline{s}\cos^2(\alpha) - s_y^2] \\ \max\{\lambda[d_y] + [s_y - Y_{max}],\ \lambda[-d_y] + [Y_{min} - s_y]\} \end{Bmatrix}$$

quadratic form : $\lambda^2 A + \lambda 2B + C$

roots : $\lambda = \dfrac{-B \pm \sqrt{B^2 - AC}}{A}$ (linear form when $A = 0$)

linear forms : $\lambda A + B$

root : $\lambda = -\dfrac{B}{A}$

The Torus of Major Axis R, Minor Axis r

$$\text{Torus}_{R,r}(\underline{l}) = \left[\underline{l}.\underline{l} - (R^2+r^2)\right]^2 + 4R^2[l_y^2 - r^2]$$

$$\text{Torus}_{R,r}(\underline{s}+\lambda\underline{d}) = \left[(\underline{s}+\lambda\underline{d}).(\underline{s}+\lambda\underline{d}) - (R^2+r^2)\right]^2 + 4R^2[(\underline{s}+\lambda\underline{d})_y^2 - r^2]$$

$$= \left[\lambda^2\underline{d}.\underline{d} + \lambda 2\underline{s}.\underline{d} + \underline{s}.\underline{s} - (R^2+r^2)\right]^2 + 4R^2[\lambda^2 d_y^2 + \lambda 2 s_y d_y + s_y^2 - r^2]$$

$$= \begin{cases} \lambda^4\ [\underline{d}.\underline{d}^2] + \\ \lambda^3\ 4[(\underline{d}.\underline{d})(\underline{s}.\underline{d})] + \\ \lambda^2\ 2[\underline{d}.\underline{d}T + 2(\underline{s}.\underline{d}^2 + R^2 d_y^2)] + \\ \lambda\ \ 4[\underline{s}.\underline{d}T + 2R^2 s_y d_y] + \\ \quad [T^2 + 4R^2(s_y^2 - r^2)]\ (\ R^2,\ r^2\ \text{constant}\ ) \end{cases}$$

where

$$T = \underline{s}.\underline{s} - [R^2+r^2]\ (\ R^2,\ r^2\ \text{constant}\ )$$

quartic form : $\lambda^4 A + \lambda^3 B + \lambda^2 C + \lambda D + E$

roots : λ found by Ferrari's solution [Korn,Korn;1968a].

Boolean Operations

Union:

$$R_{A \text{ union } B} = \{ \alpha \in R_A : Height_B(\underline{s}+\alpha\underline{d}) \geq 0 \} \cup \{ \beta \in R_B : Height_A(\underline{s}+\beta\underline{d}) \geq 0 \}$$

Intersection:

$$R_{A \text{ intersect } B} = \{ \alpha \in R_A : Height_B(\underline{s}+\alpha\underline{d}) < 0 \} \cup \{ \beta \in R_B : Height_A(\underline{s}+\beta\underline{d}) < 0 \}$$

Subtraction:

$$R_{A \text{ subtract } B} = \{ \alpha \in R_A : Height_B(\underline{s}+\alpha\underline{d}) \geq 0 \} \cup \{ \beta \in R_B : Height_A(\underline{s}+\beta\underline{d}) < 0 \}$$

Symmetric Difference:

$$R_{A \text{ difference } B} = R_A \cup R_B$$

APPENDIX D: Linear Algebra for Primitive Surface Normals

Synopsis:

Appendix D derives surface normals functions of position for a range of primitives.

Derivation

An expression for the surface normal direction at any point on a primitive is derived. This is achieved by applying the gradient operator ∇ in local space to the height polynomial of the geometry on which the local point lies. The world normal direction at a world surface point is found by a position transformation to the local surface point, evaluation of the local normal direction and a surface normal transformation back to a world surface normal [APPENDIX A]. The transformations are constant linear operations. The gradient operator ∇ acts as a linear operator on each local geometry. This is constant for all but the torus. A single compound transformation may be precalculated from world position directly to world surface normal for each geometry except the torus, for which all three transforms are applied explicitly. A unit world surface normal vector is found by appropriate scaling.

Notation

Symbol	Meaning
$\underline{c}$	World shift in world to local position transform
L	Deformation matrix for world to local position transform
$\underline{w}$	World point
$\underline{l} = L(\underline{w}-\underline{c})$	Local point
$\underline{n_l}$	Local surface normal
$S = [\underline{s_1},\underline{s_2},\underline{s_3}]$	Deformation matrix for local to world surface normal transform
$\underline{n_w}$	World surface normal
I	Deformation matrix for identity transform
$Y=\begin{bmatrix}0&0&0\\0&1&0\\0&0&0\end{bmatrix}$	Deformation matrix for projection onto local Y axis
N	Deformation matrix for world position to world surface normal transform

The Plane

$$\text{Local\_Plane}(\underline{l}) = l_y$$

$$\underline{n_l} = \nabla \text{Local\_Plane}(\underline{l}) = (0,1,0)$$

$$\underline{n_w} = S\underline{n_l} = S(0,1,0) = \underline{s_2}$$

Clipping Planes for the Cube, Cylinder and Cone

$$\text{Clipping\_Plane}_{X_{max}}(\underline{l}) = l_x - X_{max}$$

$$\underline{n_l} = \nabla \text{Clipping\_Plane}_{X_{max}}(\underline{l}) = (1,0,0)$$

$$\underline{n_w} = S\underline{n_l} = S(1,0,0) = \underline{s_1}$$

$$\text{Clipping\_Plane}_{X_{min}}(\underline{l}) = X_{min} - l_x$$

$$\underline{n_l} = \nabla \text{Clipping\_Plane}_{X_{min}}(\underline{l}) = (-1,0,0)$$

$$\underline{n_w} = S\underline{n_l} = S(-1,0,0) = -\underline{s_1}$$

$$\text{Clipping\_Plane}_{Y_{max}}(\underline{l}) = l_y - Y_{max}$$

$$\underline{n_l} = \nabla \text{Clipping\_Plane}_{Y_{max}}(\underline{l}) = (0,1,0)$$

$$\underline{n_w} = S\underline{n_l} = S(0,1,0) = \underline{s_2}$$

$$\text{Clipping\_Plane}_{Y_{min}}(\underline{l}) = Y_{min} - l_y$$

$$\underline{n_l} = \nabla \text{Clipping\_Plane}_{Y_{min}}(\underline{l}) = (0,-1,0)$$

$$\underline{n_w} = S\underline{n_l} = S(0,-1,0) = -\underline{s_2}$$

$$\text{Clipping\_Plane}_{Z_{max}}(\underline{l}) = l_z - Z_{max}$$

$$\underline{n_l} = \nabla \text{Clipping\_Plane}_{Z_{max}}(\underline{l}) = (0,0,1)$$

$$\underline{n_w} = S\underline{n_l} = S(0,0,1) = \underline{s_3}$$

$$\text{Clipping\_Plane}_{Z_{min}}(\underline{l}) = Z_{min} - l_z$$

$$\underline{n_l} = \nabla \text{Clipping\_Plane}_{Z_{min}}(\underline{l}) = (0,0,-1)$$

$$\underline{n_w} = S\underline{n_l} = S(0,0,-1) = -\underline{s_3}$$

The Sphere of Radius R

$$\text{Local\_Sphere}(\underline{l}) = l_x^2 + l_y^2 + l_z^2 - R^2$$

$$\underline{n_l} = \nabla\text{Local\_Sphere}(\underline{l}) = (2l_x, 2l_y, 2l_z) = (2I)\underline{l}$$

$$\underline{n_w} = S\underline{n_l} = S(2I)L(\underline{p}-\underline{c})$$

$\rightarrow \underline{n_w}$ is in the same direction as $N(\underline{p}-\underline{c})$ where $N = SL$

The Infinite Cylinder of Radius R

$$\text{Local\_Cylinder}(\underline{l}) = l_x^2 + l_y^2 + l_z^2 - l_y^2 - R^2$$

$$\underline{n_l} = \nabla\text{Local\_Cylinder}(\underline{l}) = (2l_x, 2l_y - 2l_y, 2_z) = 2(I-Y)\underline{l}$$

$$\underline{n_w} = S\underline{n_l} = S2(I-Y)L(\underline{p}-\underline{c})$$

$\rightarrow \underline{n_w}$ is in the same direction as $N(\underline{p}-\underline{c})$ where $N = S(I-Y)L$

The Infinite Cone of Axial Angle α

$$\text{Local\_Cone}(\underline{l}) = \cos^2(\alpha)l_x^2 + \cos^2(\alpha)l_y^2 + \cos^2(\alpha)l_z^2 - l_y^2$$

$$\underline{n_l} = \nabla\text{Local\_Cone}(\underline{l}) = (2\cos^2(\alpha)l_x, 2\cos^2(\alpha)l_y - 2l_y, 2\cos^2(\alpha)l_z) = 2(\cos^2(\alpha)I-Y)\underline{l}$$

$$\underline{n_w} = S\underline{n_l} = S2(\cos^2(\alpha)I-Y)L(\underline{p}-\underline{c})$$

$\rightarrow \underline{n_w}$ is in the same direction as $N(\underline{p}-\underline{c})$ where $N = S(\cos^2(\alpha)I-Y)L$

The Torus of Major Radius R, Minor Radius r

$$\text{Local\_Torus}(\underline{l}) = [l_x^2+l_y^2+l_z^2 - (R^2+r^2)]^2 + 4R^2[l_y^2-r^2]$$

$$\underline{n_l} = \nabla\text{Local\_Torus}(\underline{l}) = (4[\underline{l}.\underline{l}-(R^2+r^2)]l_x, 4([\underline{l}.\underline{l}-(R^2+r^2)]+2R^2)l_y, 4[\underline{l}.\underline{l}-(R^2+r^2)]l_z)$$

$$= 4([\underline{l}.\underline{l}-(R^2+r^2)]I+2R^2Y)\underline{l}$$

The $\underline{l}.\underline{l}$ term makes this matrix non-constant.

APPENDIX E: Algebra for Surface Colour Texture

Synopsis:

Appendix E derives 2D mappings from a local primitive surface point to the unit square for a range of primitives. These are texture maps for local surface colour. The texturing on each primitive is shown. Texture of the background colour is provided by a similar 2D map from an infinite sphere around the scene.

Derivation

The colour of each primitive surface is textured by local surface position according to a 2D texture map to a longitude breadth and latitude height within a unit square. Colour may be assigned procedurally as an analytic function of latitude and longitude, or by table look up according to some image scaled into the unit square. The latter provides a means of wrapping an image around a primitive surface. Procedural textures could easily be implemented to provide other texturing schemes such as Fourier texture synthesis if required. Texturing could also be generalised to a function of 3D local primitive volume rather than 2D local surface position or applied to other primitive properties such as surface normal - 'bump mapping' - or optical density rather than surface colour. However, surface colour texture by 2D table look up has proved adequate to date.

Various texture maps could be defined on each primitive. Readily visualised maps defined as continuous surjections involving minimal deformation of the unit square are preferable. They facilitate the design of a texture for a desired effect with a paint box pixel editor.

The plane primitive extends infinitely along the local X and Z axes. Continuous injective texture maps could be defined in this case. However, a necessarily high degree of stretch deformation would be involved, and as such these maps are a poor choice for general applications. A non-injective periodic function is preferable. This tiles a texture over the plane's surface. A map of low deformation with infinite range may be adapted to such a periodic function by taking the fractional part. The width or height of the tiles produced may be conveniently set through division by the appropriate length before taking this fractional part.

Notation

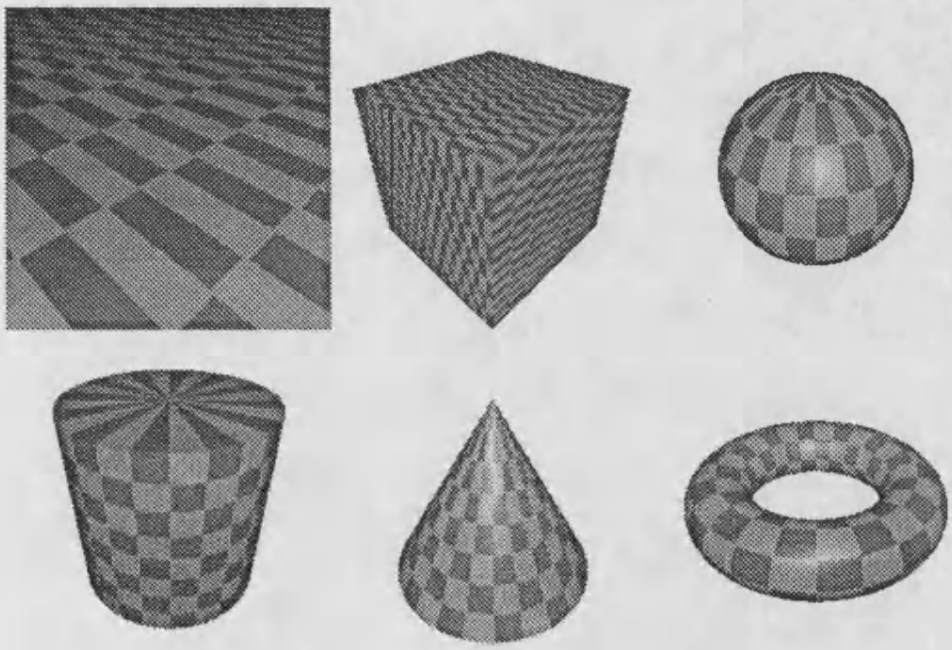
Symbol	Meaning
long $\in$ [0,1]	Local surface longitude corresponding to breadth across unit square
lat $\in$ [0,1]	Local surface latitude corresponding to height up unit square
$\underline{l}$	Local surface position
$\underline{v}$	Unit view ray direction vector
w	Tile width
h	Tile height
acos(x) $\in$ [0,π]	Inverse cosine in radians
atan2(x,y)$\in$ [-π,π]	Inverse tangent in radians
$\theta=\text{acos}\left[\frac{l_y}{\lvert\underline{l}\rvert}\right]$	Angle between local position and local y axis
$\phi=\text{atan2}(l_x,l_z)$	Angle of rotation of local position from local z axis about local y axis
fract(x) $\in$ [0,1]	Residue after subtracting the greatest smaller integer

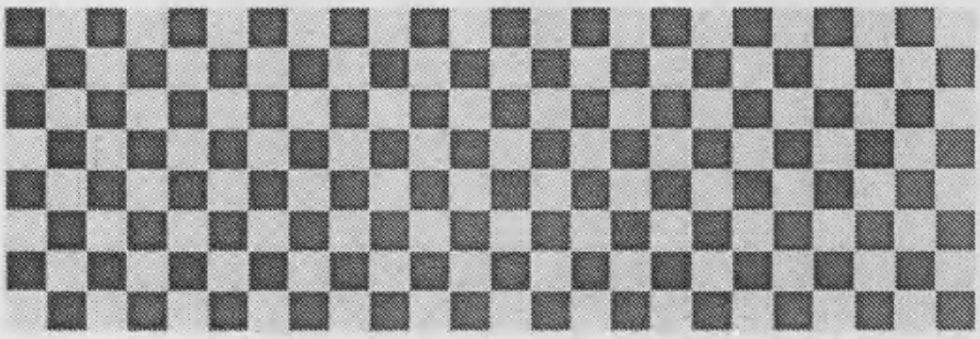
All inverse trigonometric functions map to radians. The inverse tangent function to all four quadrants is defined such that atan2 : $R^2 \rightarrow [-\pi,\pi]$ with

$$\sin(\text{atan2}(x,y)) = \frac{x}{r} \; ; \; \cos(\text{atan2}(x,y)) = \frac{y}{r} \; ; \; \text{where } r = \sqrt{x^2+y^2}.$$

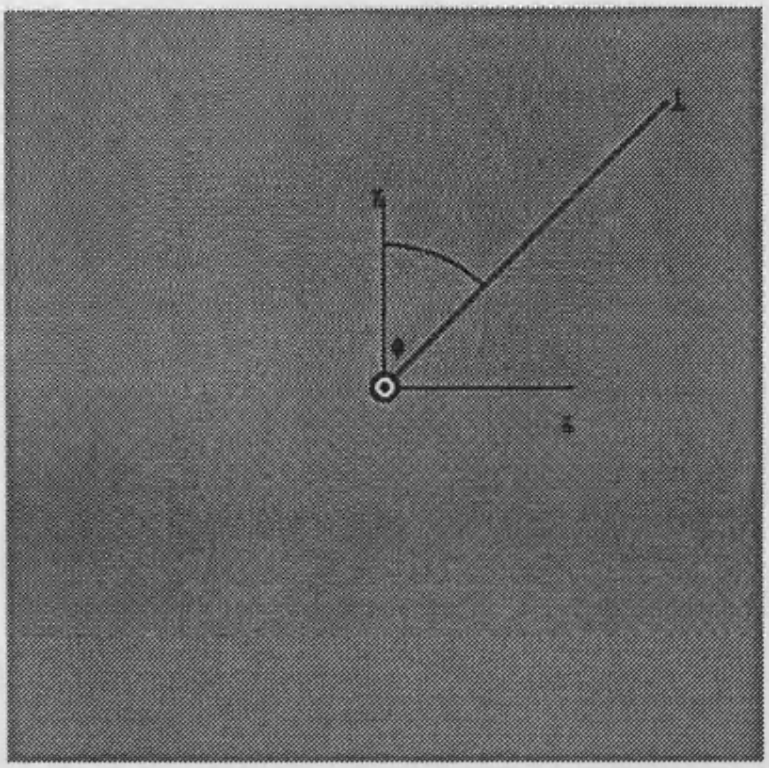
Various Primitives with a Checkered

Surface Colour Texture Map





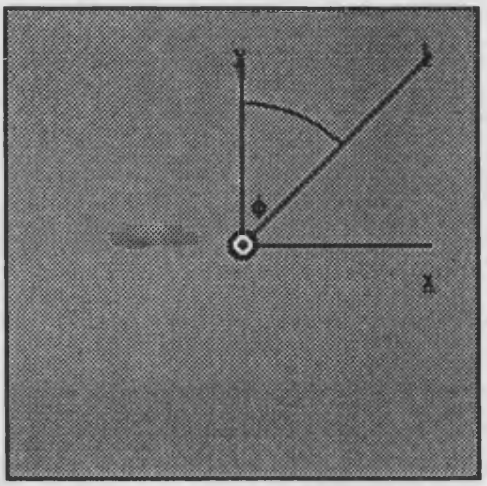
The Plane : Tile Width w, Tile Height h



$$\text{long} = \text{fract}\left[\frac{l_x}{w}\right]$$

$$\text{lat} = \text{fract}\left[\frac{l_z}{h}\right]$$

The Cube of X extent $[X_{min},X_{max}]$, Y extent $[Y_{min},Z_{max}]$, Z extent $[Z_{min},Z_{max}]$



Over clipping plane X_{max} : long = $\frac{l_z - Z_{min}}{Z_{max} - Z_{min}}$; lat = $\frac{l_y - Y_{min}}{Y_{max} - Y_{min}}$

Over clipping plane X_{min} : long = $\frac{Z_{max} - l_z}{Z_{max} - Z_{min}}$; lat = $\frac{l_y - Y_{min}}{Y_{max} - Y_{min}}$

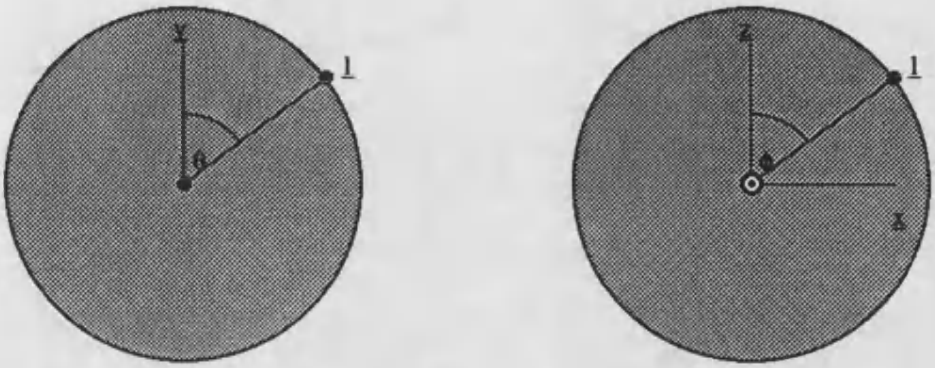
Over clipping plane Y_{max} : long = $\frac{l_x - X_{min}}{X_{max} - X_{min}}$; lat = $\frac{l_z - Z_{min}}{Z_{max} - Z_{min}}$

Over clipping plane Y_{min} : long = $\frac{X_{max} - l_x}{X_{max} - X_{min}}$; lat = $\frac{l_z - Z_{min}}{Z_{max} - Z_{min}}$

Over clipping plane Z_{max} : long = $\frac{l_y - Y_{min}}{Y_{max} - Y_{min}}$; lat = $\frac{l_x - X_{min}}{X_{max} - X_{min}}$

Over clipping plane Z_{min} : long = $\frac{Y_{max} - l_y}{Y_{max} - Y_{min}}$; lat = $\frac{l_x - X_{min}}{X_{max} - X_{min}}$

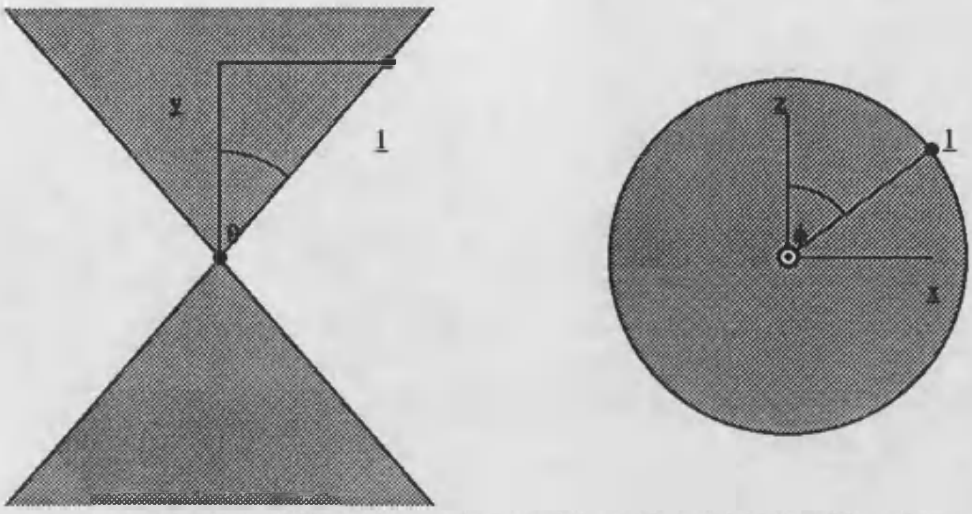
The Sphere of Radius R



$$\text{long} = \frac{\pi - \phi}{2\pi} = \frac{\pi - \text{atan2}(l_x, l_z)}{2\pi}$$

$$\text{lat} = \frac{\pi - \theta}{\pi} = \frac{\pi - \text{acos}\left[\frac{l_y}{R}\right]}{\pi}$$

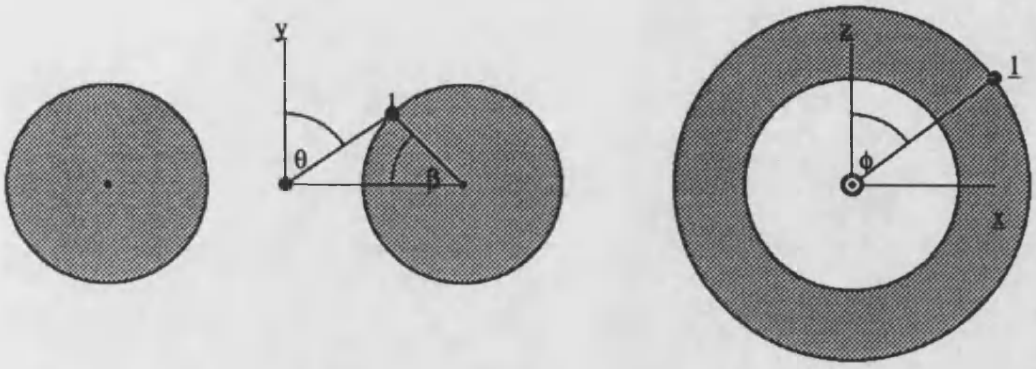
The Double Cone or Cylinder of Y extent $[Y_{min},Y_{max}]$



$$\text{long} = \frac{\pi-\phi}{2\pi} = \frac{\pi-\text{atan2}(l_x,l_z)}{2\pi}$$

$$\text{lat} = \text{fract}\left[\frac{|\underline{l}|\cos(\theta)-Y_{min}}{Y_{max}-Y_{min}}\right] = \text{fract}\left[l_y-\frac{Y_{min}}{Y_{max}-Y_{min}}\right]$$

The Torus of Major Axis R, Minor Axis r



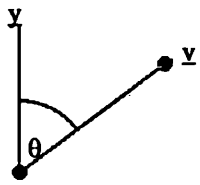
$$\text{long} = \frac{\pi - \phi}{2\pi} = \frac{\pi - \text{atan2}(l_x, l_z)}{2\pi}$$

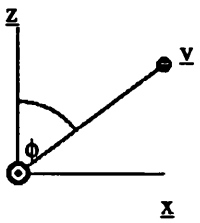
$$r\cos(\beta) = R - |\underline{l}|\sin(\theta) = R - \sqrt{|\underline{l}|^2 - |\underline{l}|^2\cos^2(\theta)} = R - \sqrt{\underline{l}.\underline{l} - l_y^2}$$

$$\text{lat} = \frac{\pi - \beta}{2\pi} = \frac{\pi - \text{atan2}(\sin(\beta), \cos(\beta))}{2\pi}$$

$$= \frac{\pi - \text{atan2}(r\sin(\beta), r\cos(\beta))}{2\pi} = \frac{\pi - \text{atan2}(l_y, R - \sqrt{\underline{l}.\underline{l} - l_y^2})}{2\pi}$$

Background Colour in View Direction (v_x , v_y , v_z)





(angles relative to world axes)

$$\text{long} = \frac{\pi - \phi}{2\pi} = \frac{\pi - \text{atan2}(v_x, v_z)}{2\pi}$$

$$\text{lat} = \frac{\pi - \theta}{\pi} = \frac{\pi - \text{acos}(v_y)}{\pi}$$

APPENDIX F: Algebra for Spawning View Rays

Synopsis:

Appendix F derives formulae for the direction of reflected and refracted view rays to be spawned in solving the view model.

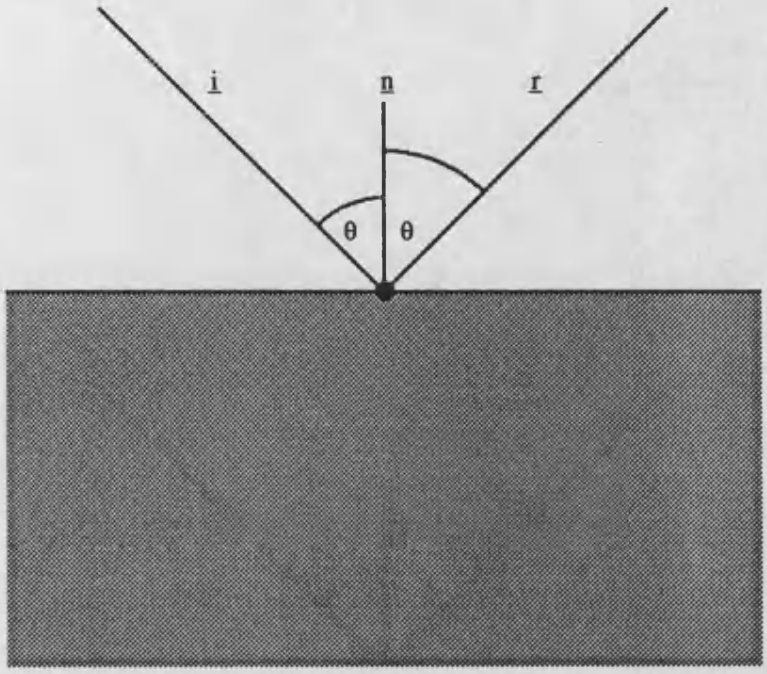
Derivation

View rays are spawned at a known surface point [APPENDIX C] of known surface normal [APPENDIX D]. The spawned rays all have source at this surface point, and their direction is found according to the laws of optical physics. Newton's law is used to find the reflected view direction, Snell's law for the refracted direction.

Notation

Symbol	Meaning
$\underline{n}$	Unit surface normal
$\underline{i}$	Incident view ray unit direction
$\underline{r}$	Spawned view ray unit direction

The Reflected View Ray



By Newton's law,

$\underline{i}$, $\underline{n}$, $\underline{r}$ are coplanar, so $\underline{r} = \alpha\underline{i} + \beta\underline{n}$

$\underline{i}\times\underline{n} = \underline{r}\times\underline{n} = \alpha\underline{i}\times\underline{n} \rightarrow \alpha = 1$

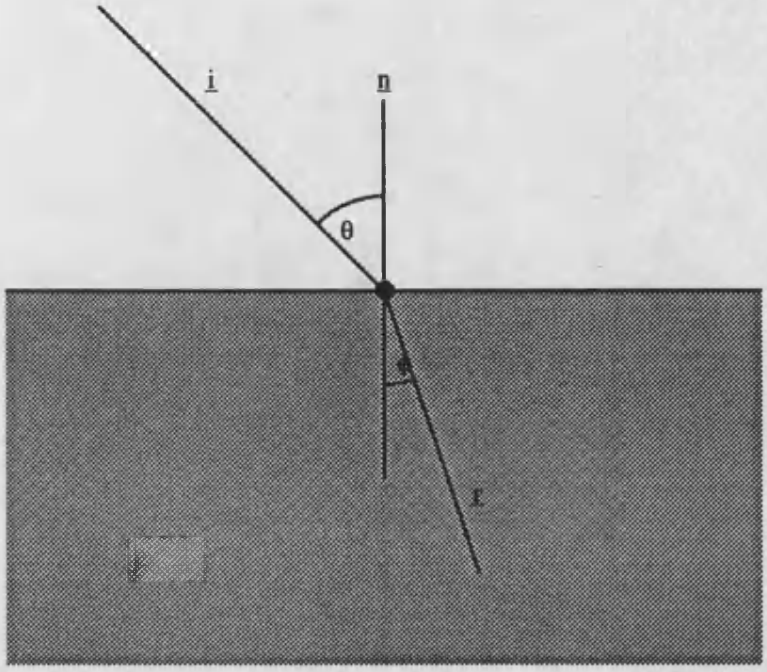
$-\underline{i}.\underline{n} = \underline{r}.\underline{n} = \alpha\underline{i}.\underline{n} + \beta \rightarrow \beta = -2\underline{i}.\underline{n}$

Therefore

$\underline{r} = \underline{i} - 2(\underline{i}.\underline{n})\underline{n}$

The Refracted View Ray

Transmission Medium of Refractive Index ν



By Snell's law,

$\underline{i}$, $\underline{n}$, $\underline{r}$ are coplanar, so $\underline{r} = \alpha\underline{i} + \beta\underline{n}$

$$\underline{i}\times\underline{n} = \nu\underline{r}\times\underline{n} = \nu\alpha\underline{i}\times\underline{n} \rightarrow \alpha = \frac{1}{\nu}$$

Moreover, since the refracted vector is of unit length

$$1 = \underline{r}.\underline{r} = \alpha^2+2\alpha\beta(\underline{i}.\underline{n})+\beta^2 \rightarrow 0 = \beta^2+2\frac{\underline{i}.\underline{n}}{\nu}\beta+\left[\frac{1}{\nu^2}-1\right] \rightarrow \beta = \frac{-\underline{i}.\underline{n}}{\nu}\pm\sqrt{\det}\ ;\ \det = \left[\frac{\underline{i}.\underline{n}}{\nu}\right]^2-\left[\frac{1}{\nu^2}-1\right]$$

Finally, for successful transmission

$$\text{sign}(\underline{i}.\underline{n}) = \text{sign}(\underline{r}.\underline{n}) = \text{sign}(\alpha\underline{i}.\underline{n}+\beta) = \text{sign}(\frac{\underline{i}.\underline{n}}{\nu}+\beta) = \text{sign}(\pm\sqrt{\det})$$ ($\det<0 \rightarrow$ total internal reflection)

Therefore,

$$\underline{r} = \frac{1}{\nu}\underline{i}+\left[-\frac{\underline{i}.\underline{n}}{\nu}\pm\sqrt{(\frac{\underline{i}.\underline{n}}{\nu})^2-(\frac{1}{\nu^2}-1)}\right]\underline{n} = \frac{1}{\nu}\left[\underline{i}+\left(-\underline{i}.\underline{n}\pm\sqrt{\underline{i}.\underline{n}^2+\nu^2-1}\right)\underline{n}\right] \quad \left[\frac{1}{\nu}\text{ , }\nu^2-1\text{ constant}\right]$$

APPENDIX G: Algebra for View Model Shading

Synopsis:

Appendix G derives formulae for the irradiance falling at given ray source point from a given ray direction, which thereby shade a visible radiant surface. Whilst full colour is to be synthesised, the formulae apply independently across the three primary light frequencies corresponding to red, green and blue, so that a monochrome image is effectively produced in each primary plane.

Derivation

The intensity of irradiance falling from a given direction is calculated as the appropriate fraction remaining from the visible radiant source after transmission through the scene environment. The radiant source will be a point on the nearest surface struck by the view ray found by solving the scene model, should such exist, or else a background environment. In the latter case the visible background colour is known [APPENDIX E]. In the former case, surface radiance is modelled by laws of optical physics according to intensity of surface irradiance, the constituent material's properties and unit vectors describing view direction, surface orientation, irradiance direction and so on. The surface point to be shaded is known from the scene model's solution for the view ray [APPENDIX C]. The surface normal is also known [APPENDIX D], as is the surface colour [APPENDIX E] and reflected and refracted view directions [APPENDIX F]. The irradiance falling on the surface point from each point light source is found by solving the scene model for a corresponding illumination ray [APPENDIX C]. Two major cases of view are modelled:

- The view ray enters the object at the surface

  when the intensity of surface radiance is summed across the five categories of diffuse, specular, ambient, reflected and transmitted;

- The view ray leaves the object at the surface

  when the intensity of surface radiance is summed across the two categories of reflected and transmitted for successful surface refraction, or only reflected for total internal reflection.

For display purposes, the view model normalises all primary colour intensities to the real range [0,1]. The intensity of surface radiance is summed with appropriate weights for an average remaining within this range.

Notation

The following notation is used throughout the view model. The model applies independently to each primary colour frequency.

Symbol	Meaning
$\underline{s}$	Irradiated ray source
$\underline{d}$	Unit ray direction vector towards radiant source
$a \in (0,\infty)$	Rate of exponential light attenuation per unit length of transmission through viewing medium
$I(\underline{s},\underline{d},a) \in [0,1]$	Irradiance falling at ray source from ray direction through attenuating medium
$\lambda \in (0,\infty]$	Path length from ray source to nearest surface point intersected by the ray (∞ if none found)
$B(\underline{d}) \in [0,1]$	Intensity of background colour in view direction; background is the radiant source when $\lambda = \infty$.
$\underline{p}$	Visible point on nearest surface; radiant source when $\lambda < \infty$.
$T(\underline{p}) \in [0,1]$	Intensity of surface colour at visible point.
$R(\underline{p},\underline{d}) \in [0,1]$	Radiance from visible point along ray direction
$R_a \in [0,1]$	Ambient radiance
$R_d \in [0,1]$	Diffuse radiance
$R_t \in [0,1]$	Transmitted (refracted) radiance
$R_s \in [0,1]$	Specular radiance
$R_r \in [0,1]$	Reflected radiance

Symbol	Meaning
$f_m \in [0,1]$	Fraction of *mirrored* specular and reflected radiance in surface radiance.
$1-f_m \in [0,1]$	Remaining fraction of *non-mirrored* ambient, diffuse and transmitted radiance in surface radiance.
$f_r \in [0,1]$	Fraction of *reflected* view radiance in mirrored radiance
$1-f_r \in [0,1]$	Remaining fraction of specular highlight radiance in mirrored radiance
$f_t \in [0,1]$	Fraction of *transmitted* view radiance in non-mirrored radiance
$1-f_t \in [0,1]$	Remaining fraction of ambient and diffuse radiance in non-mirrored radiance
$f_a \in [0,1]$	Fraction of *ambient* radiance in ambient and diffuse radiance
$1-f_a \in [0,1]$	Remaining fraction of diffuse radiance in ambient and diffuse radiance
$s \in [0,\infty)$	Sharpness of specular highlights, modelling variance of microfacet normal distribution
$a_m \in [0,\infty]$	Rate of exponential light attenuation per unit length of transmission through constituent material of surface object
$\underline{n}$	Unit surface normal vector at visible point
$\underline{r}$	Vector in reflected view direction
$\underline{t}$	Vector in transmitted (refracted) view direction
c	Light source count
$L_i \in [0,1]$	Irradiant light intensity from i^{th} light source
$\underline{l_i}$	Unit vector from surface point to i^{th} light source

No visible surface point

Characterisation : $\lambda = \infty$

$$I(\underline{s},\underline{d},a) = \begin{cases} \text{if } (a>0) \text{ then} \\ \quad 0 \\ \text{else } (a=0) \text{ then} \\ \quad B(\underline{d}) \end{cases}$$

Visible surface point

Characterisation : $\lambda < \infty$

$$I(\underline{s},\underline{d},a) = e^{-\lambda s}R(\underline{p},\underline{d})$$

where $R(\underline{p},\underline{d})$ is defined as follows:

Ray leaves (transparent) object at visible surface point

Characterisation : $\underline{d}.\underline{n} > 0$

$$R(\underline{p},\underline{d}) = \begin{cases} \text{if (successful refraction) then} \\ \quad f_m I(\underline{p},\underline{r},a)+(1-f_m)I(\underline{p},\underline{t},a-a_m) \\ \text{else (total internal reflection) then} \\ \quad I(\underline{p},\underline{r},a) \end{cases}$$

Ray enters object at visible surface point

Characterisation : $\underline{d}.\underline{n} < 0$

$$R(\underline{p},\underline{d}) = (1-f_m)\left\{(1-f_t)(f_a R_a+(1-f_a)R_d)+f_t R_t\right\}+f_m\left\{(1-f_r)R_s+f_r R_r\right\}$$

Where

$$\left.\right\}+f_m\left\{(1-f_r)R_s+f_r R_r\right\}$$

$R_a = T(\underline{p})$

$R_d = T(\underline{p})\sum_{i=1}^{c} L_i \max(\underline{l}_i.\underline{n},0)$ (Lambert's Law)

$R_t = T(\underline{p})I(\underline{p},\underline{t},a+a_m)$ (Snell's Law)

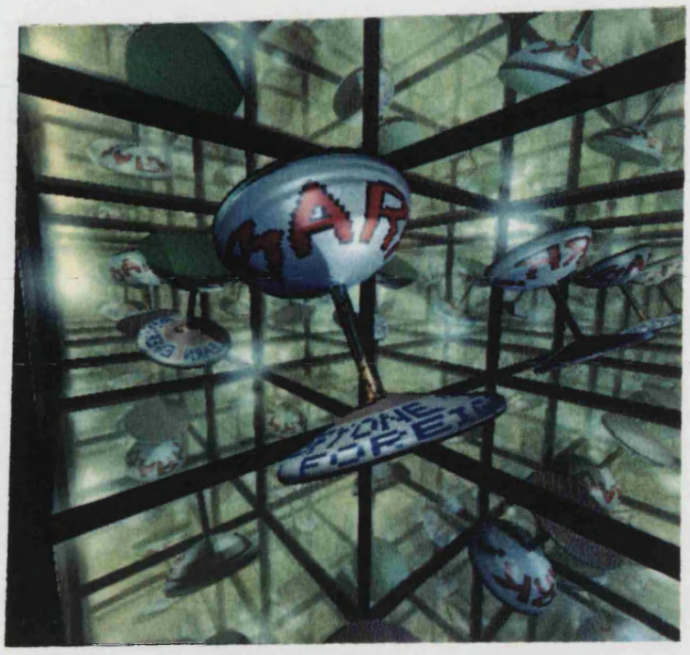
$R_s = \sum_{i=1}^{c} L_i \max(\underline{l}_i.\underline{r},0)^s$ (Phong's Model)

$R_r = I(\underline{p},\underline{r},a)$ (Newton's Law)

APPENDIX H: Gallery of Specimen Images

Synopsis:

Appendix H constitutes a gallery of specimen images demonstrating the realism synthesised by ray tracing. All the images are produced from implementations developed during this research.

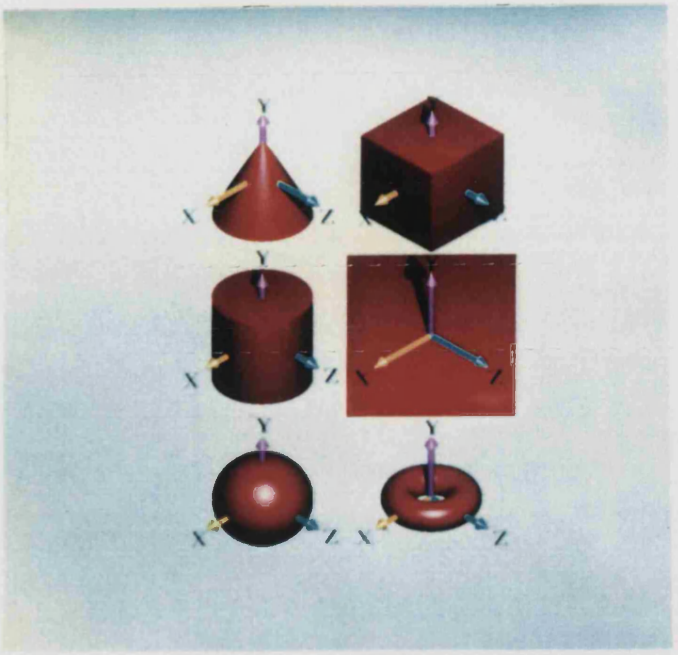


1: A Goblet in A Box of Mirrors

2: A Refractive Goblet

3: A CSG Model of a Refractive Dice

4: A Pyramid of Dice

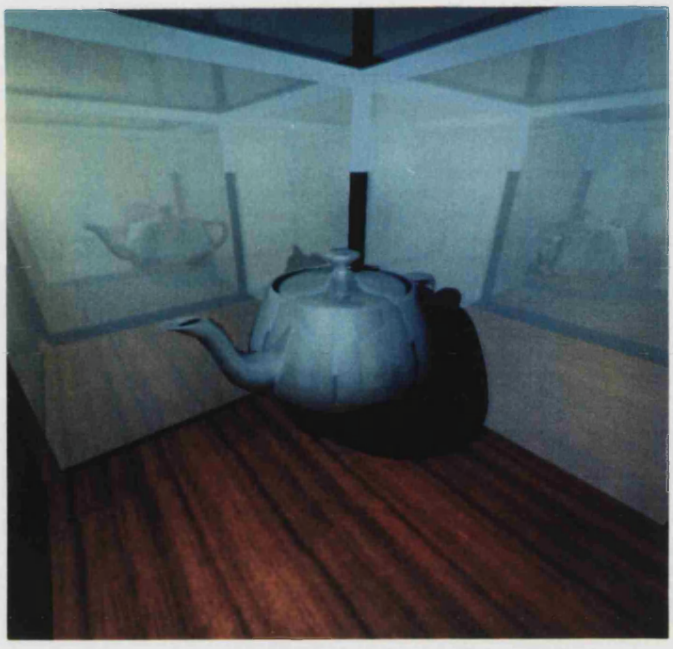


5: Primitive Solids - The Cone, Cube, Cylinder, Plane, Sphere and Torus

6: A CSG model of a Mug [Fig 2.2.1a]

7: A Smooth-Shaded Polyhedral Model of a Teapot

8: A Reflective Teapot

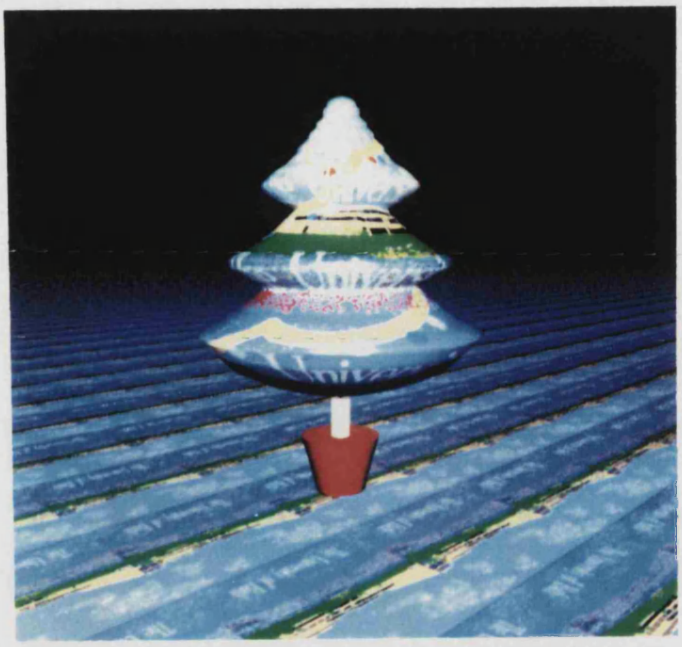


9: A Teapot in a Box of Mirrors

10: A Recursive Arrangement of Reflective Spheres

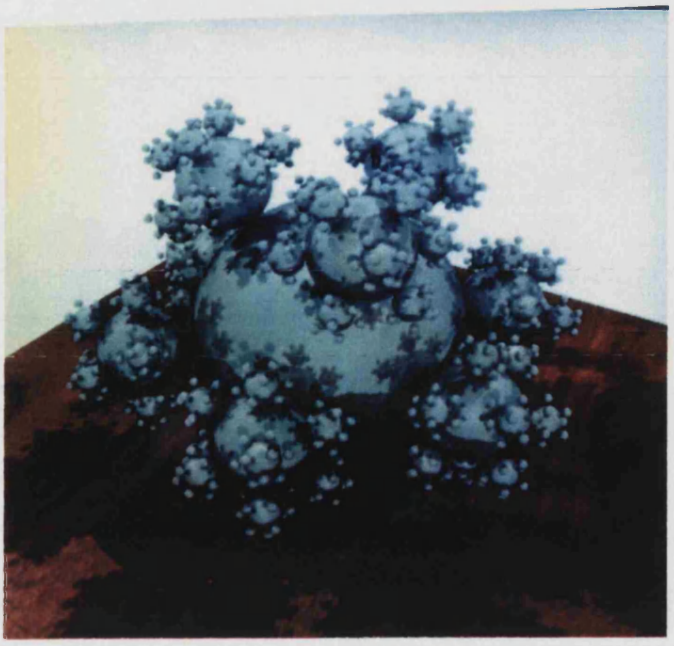
11: A Stair-Case

12: A Scene Comprising Over Twenty Thousand Objects

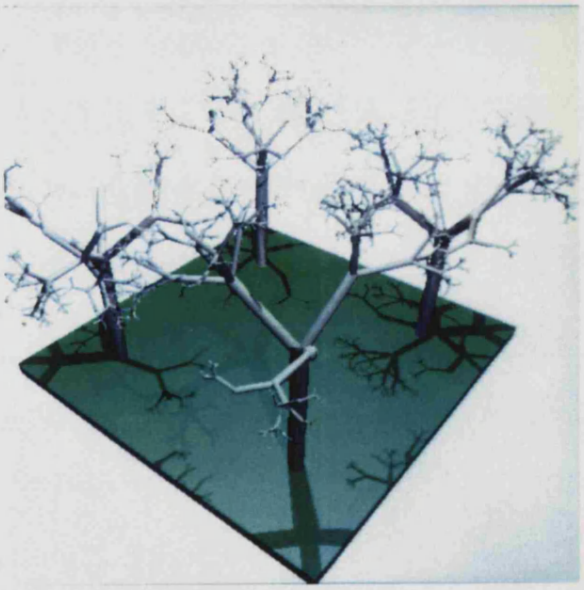


13: A Tree Modelled with Truncated Cones

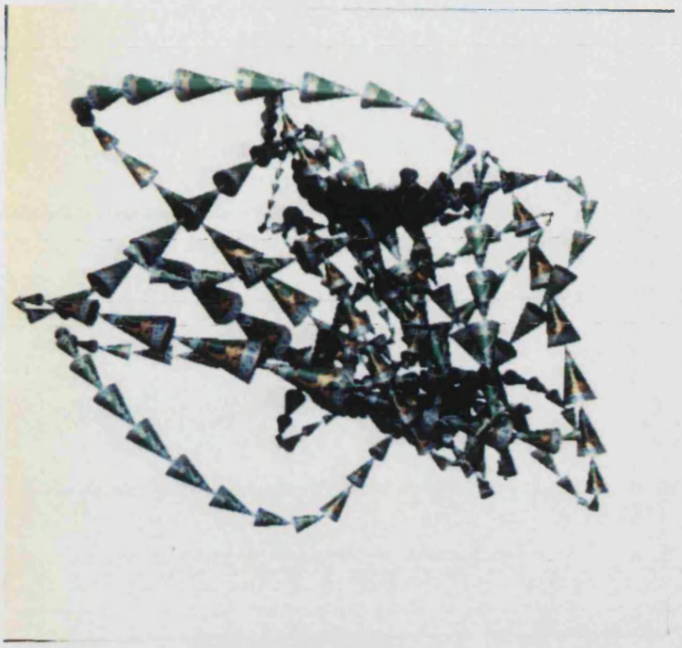
14: A Tree in a Box of Mirrors



17: A Recursive Arrangement of Reflective Spheres [Fig 7.4a]

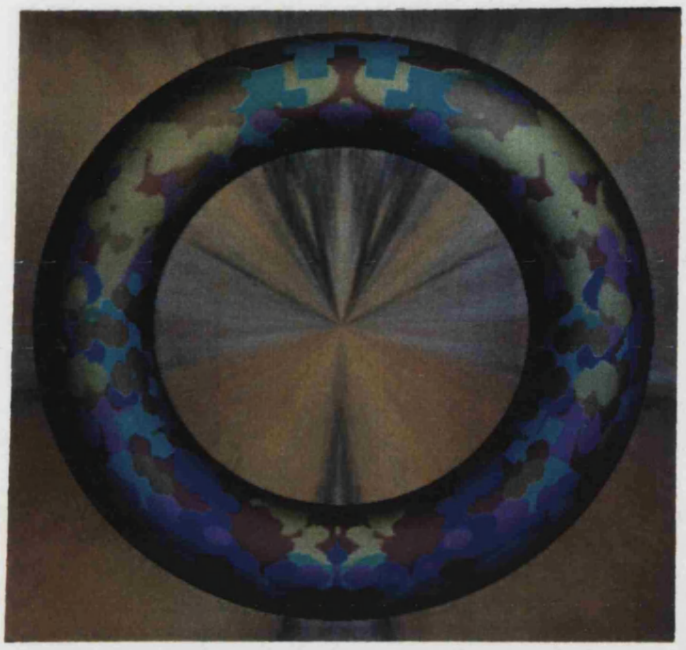


18: Trees Modelled with Cylinders and Spheres [Fig 7.4b]

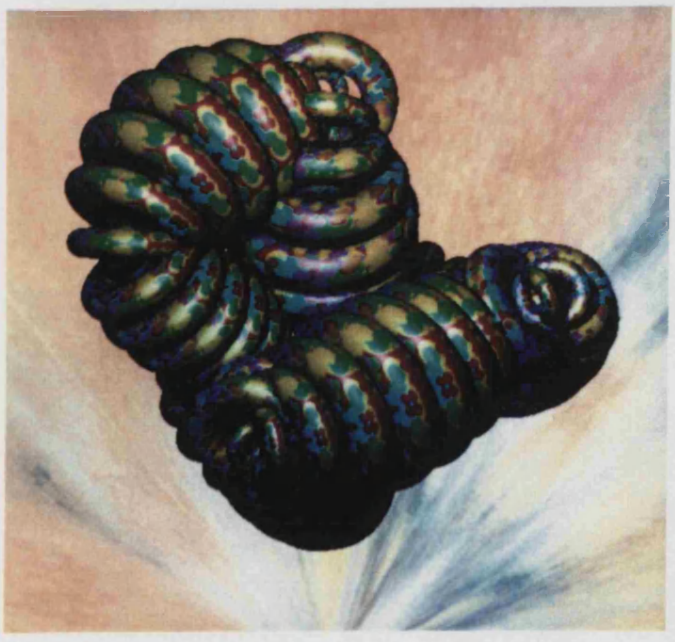


19: An Arrangement of Cones along a 3D Spline Path [Fig 7.4c]

20: Several Robots Inside a Room [Fig 7.4d]

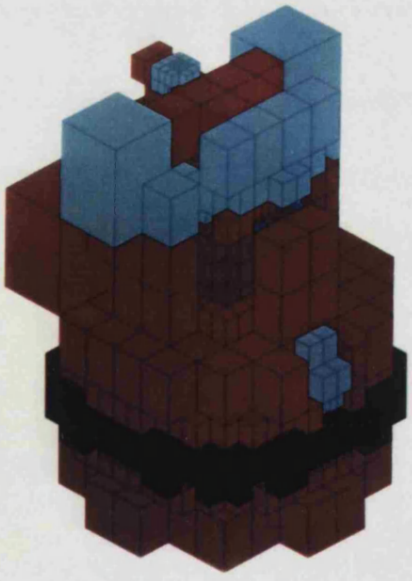


21: A Single Torus [Fig 7.7.2b]

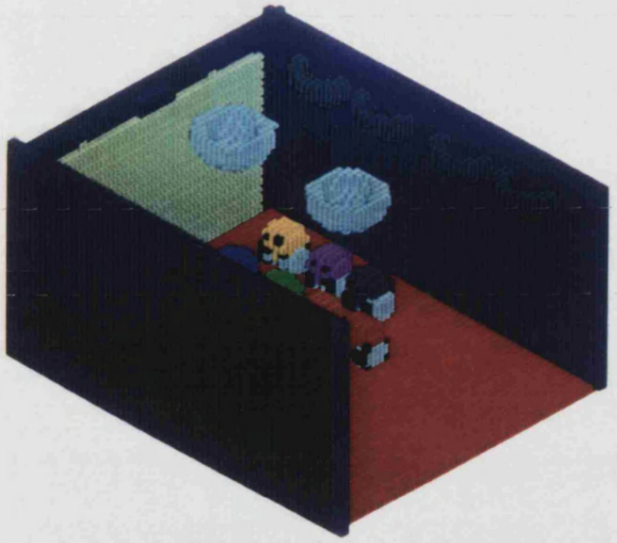


22: Eighty Tori [Fig 7.7.2c]

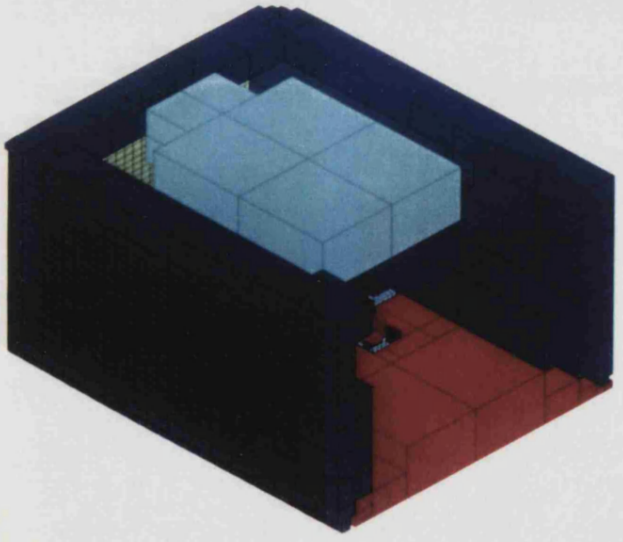
23: An Octtree Decomposition of *Simplicity* 0 [Fig 6.1a]



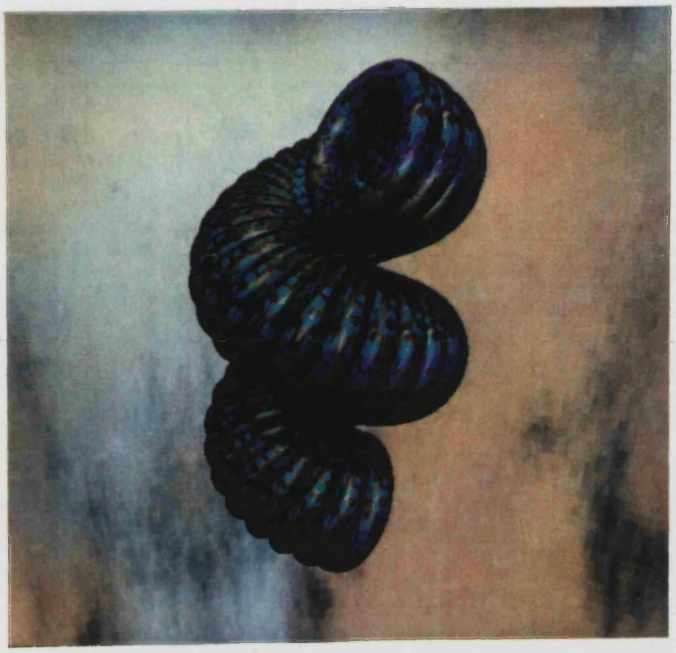
24: An Octtree Decomposition of *Simplicity* 1 [Fig 6.1a]



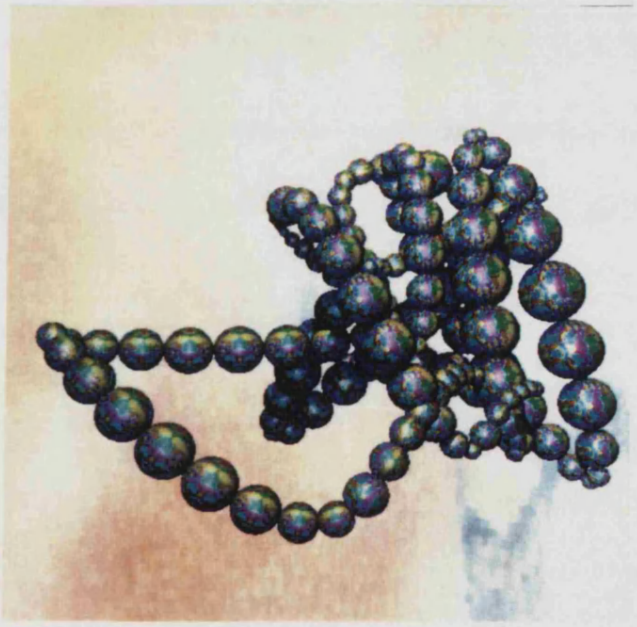
25: An Octtree Decomposition of *Simplicity* 0 [Fig 6.1a]



26: An Octtree Decomposition of *Simplicity* 1 [Fig 6.1a]

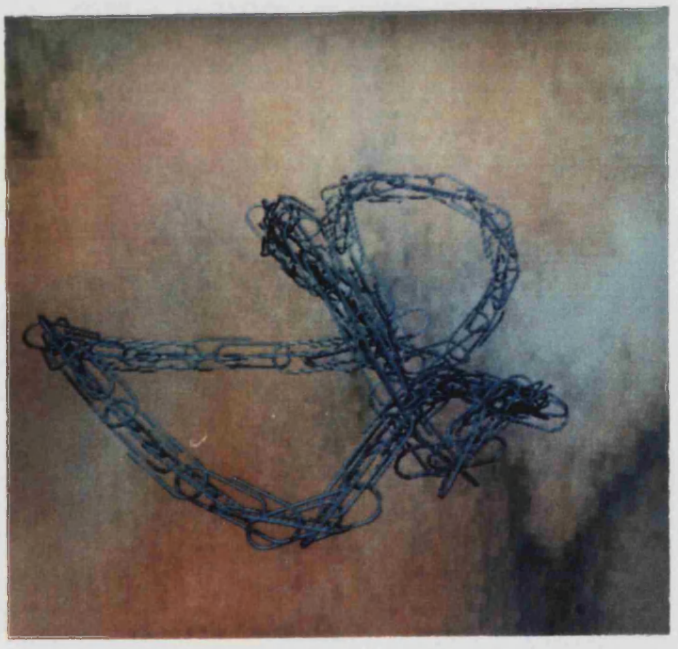


27: A Helix of Tori

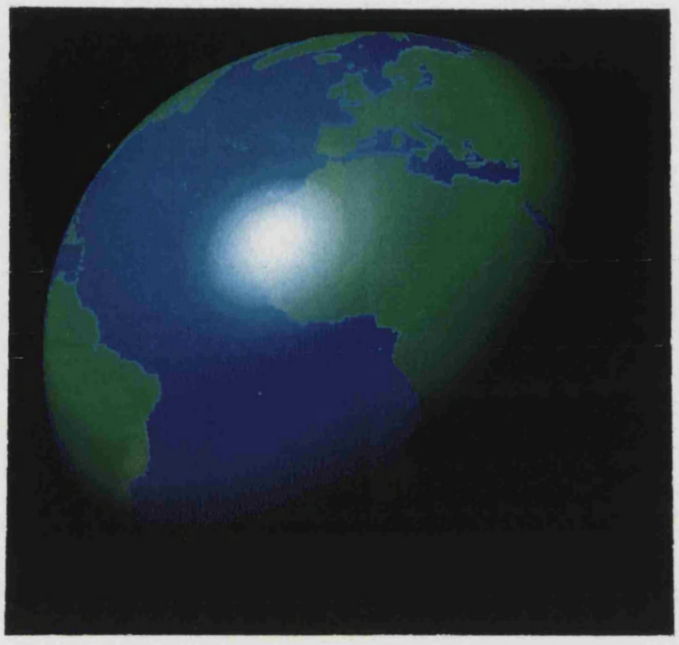


28: An Arrangement of Spheres along a 3D Spline Path

29: A CSG Model of Paper Clip Built from Cylinders, Clipping Planes and Tori

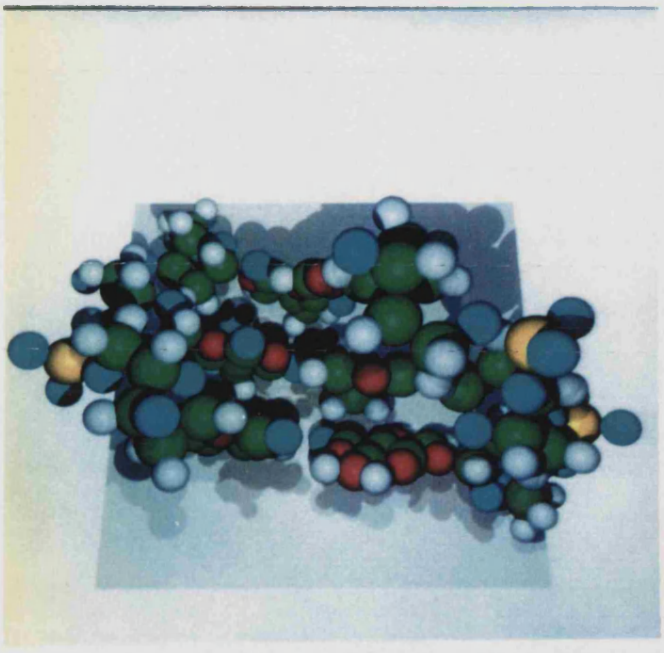


30: An Arrangement of Interlocking Paper Clips along a 3D Spline Path

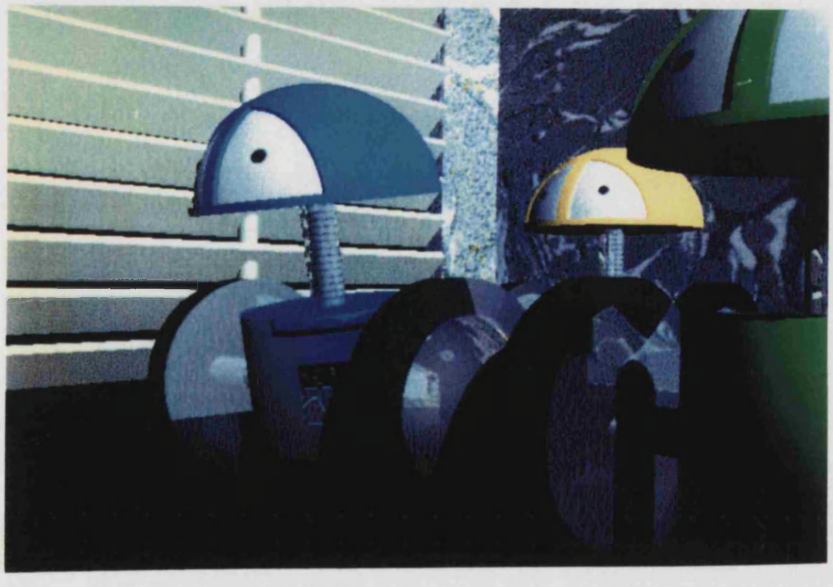


31: A Single Texture Mapped Sphere

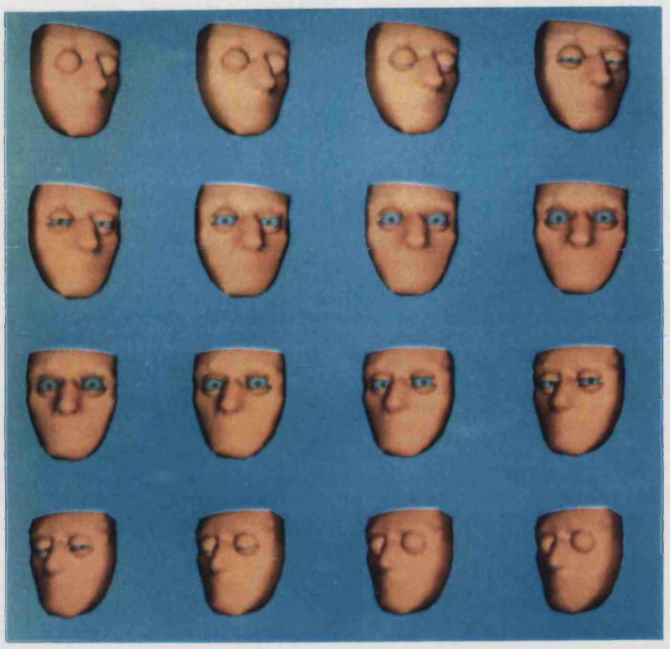
32: Numerous Texture Mapped Spheres



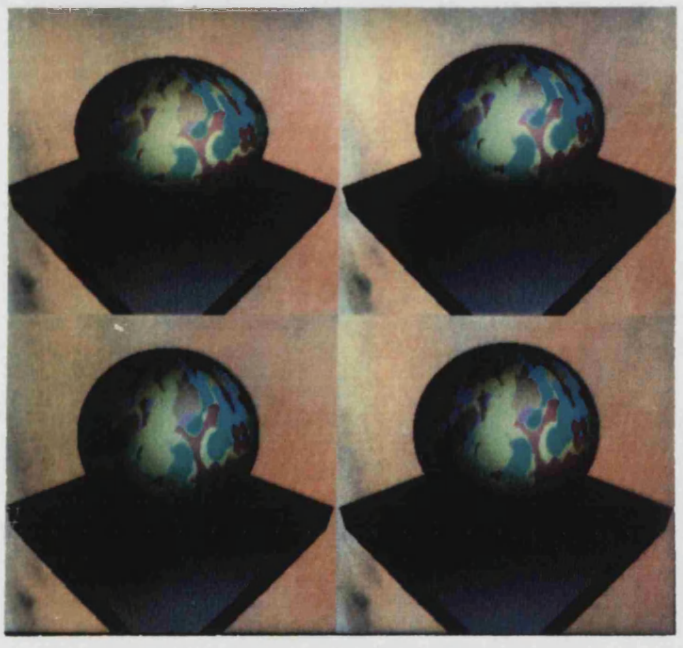
33: A Molecule Modelled with Spheres



34: A Frame from an Animated Film [John;1989]



35: Frames from an Animated Sequence of a Blinking Face



36: Frames from an Animated Sequence of a Bouncing, Deforming Ball

BIBLIOGRAPHY

Amanatides; 1984:
Ray Tracing with Cones *SIGGRAPH ACM* pp.129-135

Amanatides; 1987:
Realism in Computer Graphics: A Survey *IEEE CG&A January* pp.44-56

Appel; 1970:
Some Techniques for Shading Machine Renderings of Solids *Proc. AFIPS JSCC* pp.37-45

Avro, Kirk; 1987:
Fast Ray Tracing by Ray Classification *SIGGRAPH ACM* pp.55-64

Beacon, Dodsworth, Howe, Oliver, Saia; 1989:
Boundary Evaluation Using Inner and Outer Sets: The ISOS Method *IEEE CG&A March* pp.39-51

Bresenham; 1965:
Algorithm for Computer Control of a Digital Plotter *IBM Systems Journal* pp.25-30

Burr; 1986:
Ray Tracing Deformed Surfaces *SIGGRAPH ACM* pp.287-296

Catmull; 1978:
A Hidden-Surface Algorithm with Anti-Aliasing *Computer Graphics August*

Cook, Porter, Carpenter; 1984:
Distributed Ray Tracing *SIGGRAPH ACM* pp.137-145

Dippe, Swensen; 1984:
An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis *SIGGRAPH ACM* pp.149-158

Fabbrinni, Montani; 1986:
Autumnal Quadtrees *The Computer Journal May* pp.474

Foley, Van Dam; 1984:
Fundamentals of Interactive Computer Graphics pp.146

Frenkel; 1989:
Volume Rendering *Comm. ACM* pp.426-435

Fujimoto, Tanaka, Iwata; 1986:
ARTS: Accelerated Ray-Tracing System *IEEE CG&A April* pp.16-26

Gargantini; 1982:
An Effective Way to Represent Quadtrees *Comm. of the ACM December* pp.905-910

Glassner; 1984:
Space Subdivision for Fast Ray Tracing *IEEE CG&A October* pp.15-22

Glassner; 1988:
Spacetime Ray Tracing for Animation *IEEE CG&A March* pp.60-70

Goldsmith, Salmon; 1987:
Automatic Creation of Object Hierarchies for Ray Tracing *IEEE CG&A May* pp.14-20

Gouraud; 1971:
Computer Display of Curved Surfaces *IEEE Transactions June* pp.623

Haines, Greenberg; 1986:
The Light Buffer: A Shadow-Testing Accelerator *IEEE CG&A September* pp.6-16

Haines; 1987:
A Proposal for Standard Graphics Environments *IEEE CG&A November* pp.3-5

Haines; 1988:
The Suitability of Octtrees for Accelerated Ray Tracing *Ray Tracing News (e-mail edition) Issues 1-3 January-March*

Heckbert, Hanrahan; 1984:
Beam Tracing Polygonal Objects *SIGGRAPH ACM* pp.119-127

Huffman; 1952:
A Method for the Construction of Minimum-Redundancy Codes *Proc. IRE* pp.1098-1101

Hunter, Steiglitz; 1980:
Operations on Images Using Quad Trees *IEEE Trans. Pattern Analysis and Machine Intelligence January* pp.27-35

Jackins, Tanimoto; 1980:
Octtrees and Their Use in Representing Three-Dimensional Objects *Computer Graphics and Image Processing November* pp.249-270

John; 1989:
Animated Film: 'Hit the Show, MAC!' *The University of Bath*

Joy, Bhetanabhotla; 1986:
Ray Tracing Parametric Surface Patches Utilizing Numerical Techniques and Ray Coherence *SIGGRAPH ACM* pp.279-285

Kajiya, Von Herzen; 1984:
Ray Tracing Volume Densities *SIGGRAPH ACM* pp.165-174

Kaplan; 1985:
Space-Tracing, A Constant Time Ray-Tracer *SIGGRAPH ACM Tutorial July*

Kay, Kajiya; 1986:
Ray Tracing Complex Scenes *SIGGRAPH ACM* pp.269-278

Korn, Korn; 1968a:
Roots of Univariate Quadratics, Cubics and Quartics *Mathematical Handbook for Scientists and Engineers, Second Edition* pp.22-24

Korn, Korn; 1968b:
Sturm's Method *Mathematical Handbook for Scientists and Engineers, Second Edition* pp.18

Lathrop; 1988:
Lazy Octtree Construction *Ray Tracing News (e-mail edition) Issue 1 January*

Marsh; 1987:
UgRay: An Efficient Ray-Tracing Renderer for UniGrafix *Master's Project Report, University of California*

Meagher; 1982:
Geometric Modelling Using Octtree Encoding *Computer Graphics and Image Processing June* pp.129-147

Moore; 1965:
The Automatic Analysis and Control of Error in Digital Computation based on the Use of Interval Numbers *Error in Digital Computation, Vol.1* pp.61-130

Moore; 1979:
Methods and Applications of Interval Analysis *SIAM Studies in Applied Mathematics*

Morton; 1966:
A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing *IBM Ltd, Ottawa, Canada*

Muller; 1987:
Parallel Algorithms for Realistic Image Synthesis *Department of Computer Science, University of Karlsruhe, West Germany*

Myers; 1982:
An Industrial Perspective on Solid Modelling *IEEE CG&A March* pp.86-97

Patel; 1989:
Computer Animation of Faces *The University of Bath*

Phong; 1975:
Illumination for Computer-Generated Picture *CACM June* pp.311-317

Plunkett, Bailey; 1985:
The Vectorization of a Ray-Tracing Algorithm for Improved Execution Speed *IEEE CG&A August* pp.52-60

Rall; 1981:
Automatic Differentiation: Techniques and Applications *Lecture Notes in Computer Science* pp.113-121

Roth; 1982:
Ray Casting for Modelling Solids *Computer Graphics and Image Processing Vol.18* pp.109-144

Rubin, Whitted; 1980:
A 3-Dimensional Representation for Fast Rendering of Complex Scenes *Computer Graphics July* pp.110-116

Samet, Webber; 1988:
Hierarchal Data Structures and Algorithms for Computer Graphics *IEEE CG&A May & July* pp.44-68 & 59-75

Sederberg, Anderson; 1984:
Ray Tracing of Steiner Patches *SIGGRAPH ACM* pp.159-268

Spackman; 1987:
Third Six Monthly PhD Research Report *The University of Bath* pp.19

Sweeny, Bartels; 1986:
Ray Tracing Free-Form B-Spline Surfaces *IEEE CG&A February* pp.41-49

Waij, Heckbert, Glassner; 1988:
Octtree Representations of Quadric Surfaces *Ray Tracing News (e-mail edition) Issue 2 March*

Whitted; 1980:
An Improved Illumination Model for Shaded Display *Comm. ACM June* pp.343-349

Williams, Buxton, Buxton; 1986:
Distributed Ray Tracing Using an SIMD Processor Array *GEC Hirst Research Centre*

Woodwark, Bowyer; 1986:
Better and Faster Pictures from Solid Models *Computer-Aided Engineering Journal February* pp.17-24

Wyvill, Kunii, Shirai; 1986:
Space Division for Ray Tracing in CSG *IEEE CG&A April* pp.28-34

Zhang, Bowyer; 1986:
CSG Set-Theoretic Solid Modelling and NC Machining of Blend Surfaces *Proc. 2nd ACM Symposium on Computational Geometry June* pp.236-245